Model-based Adaptation of Behavioural Mismatching Components

Carlos Canal, Pascal Poizat, Gwen Salaün

Model-Based Adaptation of Behavioural Mismatching Components

Carlos Canal, Pascal Poizat, and Gwen Salaün

C. Canal is with LCC, Universidad de Málaga, Spain.

P. Poizat is with IBISC FRE 2873 CNRS, Université d'Évry Val d'Essonne and ARLES project, INRIA, France.

G. Salaün is with LCC, Universidad de Málaga, Spain.

Abstract

Component-Based Software Engineering focuses on the reuse of existing software components. In practice, most components cannot be integrated directly into an application-to-be, because they are incompatible. Software Adaptation aims at generating, as automatically as possible, adaptors to compensate mismatch between component interfaces, and is therefore a promising solution for the development of a real market of components promoting software reuse. In this article, we present our approach for software adaptation which relies on an abstract notation based on synchronous vectors and transition systems for governing adaptation rules. Our proposal is supported by dedicated algorithms that generate automatically adaptor protocols. These algorithms have been implemented in a tool, called Adaptor, that can be used through a user-friendly graphical interface.

Index Terms

Software components, interfaces, mismatch, composition, software adaptation, adaptation contracts, vectors, transition systems, synchronous products, Petri nets, tools.

I. Introduction

Component-Based Software Engineering (CBSE) aims at building new systems by assembling existing software components, which would jointly realize the system desired functionality. However, one of the main issues raised by this paradigm is that in practice we cannot expect that any given software component perfectly matches the needs of a system where it is trying to be reused, nor that the components being assembled fit perfectly one another. Reusing software often requires a certain degree of adaptation [1], [2], especially in presence of legacy code. To deal with these problems, *Software Adaptation* [3], [4] is emerging as a new discipline, concerned with providing techniques to arrange already developed pieces of software in order to reuse them in new systems, accommodating the potential mismatches arising from their composition.

Software Adaptation promotes the use of *adaptors*, specific computational entities developed for guaranteeing that a set of mismatching components will interact correctly. Software adaptation is different from software evolution, component customization, or adaptive middleware. *Software evolution* aims at modifying the code of the components, for instance to take a new functionality into account, whereas adaptation works in a non-intrusive way, that is without modifying the code of the components, which is important due to their black-box nature. In the case of *customization*, the end-user may adjust the component behaviour by tuning a fixed set of component parameters, which have been considered and defined at design time by the developer. Finally, dedicated *adaptive middleware* [5] can be used to put the adaptation process into action, once an adaptor model has been obtained. In this sense, adaptive middleware complements software adaptation, which deals with adaptor modeling and synthesis, providing the means for the actual implementation of the proposal.

CBSE postulates that a component must be reusable from its interface [6], which in fact constitutes its full technical specification. The characteristics and expressiveness of the language used for interface description determines the degree of interoperability we can achieve using it, and the kind of problems that can be solved. We distinguish several levels of interoperability, and accordingly of interface description [2], [4], [7]: *technical* level (data encoding and

framework-related issues), *signature* level (operation names and types), *behavioural* level (interaction *protocols*), *quality of service* level (non-functional properties such as security or efficiency), and *semantic* level sometimes referred as conceptual level (functional specification of what the component actually does). At each one, mismatch may occur and have to be corrected. Currently, industrial component models, by using Interface Description Languages (IDLs), are able to solve most of the technical interaction problems, but they fail to address mismatch at the higher levels. Numerous approaches have been presented for extending component interfaces with protocols (see, for instance, [8]–[13]) thus resulting in what we call Behavioural IDLs (BIDLs). This interoperability level is essential because, even if components match from a signature point of view, their combination can lead to erroneous behaviours or deadlock situations if the designer is not aware of their execution flows, and does not take them into account while building the full system.

In this article, we propose a model-based adaptation approach focusing on mismatch appearing at the behavioural level. Yet, since the component protocols are based on message exchange relative to the component operations, we also address name mismatch at the signature level. The approach (see Fig. 1 for a graphical overview of it) takes as input the behavioural interfaces of components to be adapted, and an adaptation *contract* [4], that is an abstract description of the constraints which must be respected to make the involved components work together. Given these two elements, an adaptor protocol is generated in an automatic way.

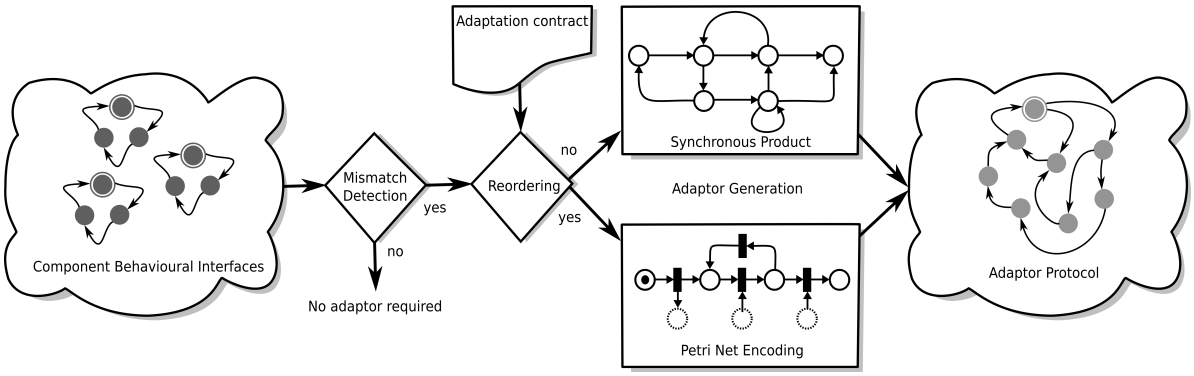


Fig. 1. Overview of our model-based adaptation approach

The adaptation process begins with two (or more) components that are not able —as they are— to interact successfully (*i.e.*, ending in correct termination states). To compensate such mismatch, we propose to use synchronous vectors as adaptation contract language to make explicit the interactions between components, possibly on different message names. Our notation also allows the specification of ordering constraints on interactions, which enables one to describe in an abstract way more complex adaptation scenarios. In order to generate adaptor protocols for such contracts, we present in this article two algorithms that automate the adaptation process. The first one is based on synchronous products, and the second one is based on Petri net encodings. Compared to the former, the latter induces a higher computational complexity, but is able to reorder messages when necessary, and then

ensures a correct interaction when several components have the messages exchanged in their protocols which are not ordered correspondingly. Reordering is worked out desynchronising the message emission by one component and the message reception in another one. When required, emitted messages are temporarily memorised until they are used for effective interaction. This is why a formalism capable of representing memory, such as Petri nets, is required. The adaptation techniques we present in this article have been implemented in a tool, called Adaptor, which has been applied to many non-trivial examples, *i.e.*, examples where adaptor protocols could not have been obtained by hand.

A preliminary version of this work has been presented in [14], and is extended here in several aspects: (i) introduction to the Petri nets concepts used in our proposal, (ii) detailed descriptions and proofs of the adaptation algorithms, (iii) presentation of the Adaptor tool, (iv) illustration on a more realistic and bigger case study from the pervasive computing domain, and (v) an updated review and comparison with related work.

The remainder of the article is organised as follows. Section II formally introduces our component interface model, and defines interface mismatch. Section III focuses on the adaptation contract notation. Section IV presents a first approach to component adaptation based on synchronous products. Section V presents a second solution which goes further, considering reordering through the encoding of contracts and behavioural interfaces into Petri nets. Section VI gives an overview of the Adaptor tool. In Section VII, we survey the more advanced proposals for behavioural software adaptation, and compare to them. Finally, Section VIII ends the article with some concluding remarks.

II. Interfaces and Mismatch

In this section, we present first the model of interfaces through which components are accessed and used. Then, we define the notion of interface mismatch that our approach addresses.

A. Component Interfaces

We assume that component interfaces are given using both a signature and a behavioural interface. Signature interfaces usually correspond in component-based frameworks (*e.g.*, CCM, .NET or J2EE) to operation profiles described using an IDL, *i.e.*, operation names associated with argument and return types relative to the data being exchanged when the operation is called. Since we focus on the behavioural level in this article, we omit in the signature interfaces the elements relative to data exchange. This means that a signature is taken as a disjoint set of provided and required operation names. Such abstractions from data exchange are often used in software engineering, *e.g.*, to check interface compatibility [11] or to perform component verification [10], [12]. Additionally, we propose that behavioural interfaces are represented by means of Labelled Transition Systems (LTSs). Message-based communication between components is therefore represented using *events* relative to the emission (denoted using !) and reception (denoted using ?) of *messages* corresponding to operation calls.

However, taking data exchange into account is important to ensure full compatibility. So far, this can be supported in our approach using additional messages as follows. The emission by a component of a message login with two data

information, username and password, would be encoded by the sequence of events login!.username!.password! in the component LTS. Accordingly, the reception in a component of a message login with two data information, username and password, would be encoded by the sequence of events login?.username?.password? in the component LTS. Provided this encoding is performed as a pre-processing, and the adaptation contract takes the additional messages into account, the protocols can be adapted, as demonstrated in [15] where we have applied our adaptation techniques to Windows Workflow Foundation (WF) [16] which belongs to the .NET Framework 3.0 developed by Microsoft®. Related perspectives are further discussed in Section VIII.

Definition 1 (LTS): A *Labelled Transition System* is a tuple (A, S, I, F, T) where: A is an alphabet (set of events), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function.

Final states correspond to correct service terminations in components. To support the correctness of the adaptation process, we further assume that the initial state is also final ($I \in F$). The alphabet of the LTS is built on the component signature. This means that for each provided operation p in the signature, there is a message p and an event $p?$ in the alphabet, and, for each required operation r, there is a message r and an event $r!$ in the alphabet. Complementary events are denoted with the same name of message and opposite directions. Consequently, the complementing function on events is defined as: $\overline{e?} = e!$, and $\overline{e!} = e?$.

LTSs are adequate models as far as user-friendliness and development of formal algorithms are concerned. However, higher-level languages such as process algebras [17] can be used to define behavioural interfaces in a more concise way. In a former version of this work [14], the sequential subset of CCS [18] was used as BIDL. Moreover, CCS descriptions of component behavioural interfaces can be easily translated into LTS models using the operational rules defining the semantics of the formalism. In this article, since we focus on the adaptor model generation, we only present and work using LTS models. In [15], the reader will find more details of how LTSs can be extracted from component languages (namely, in this work, the Windows Workflow Foundation language), and how an adaptor model can be transformed into a component language program.

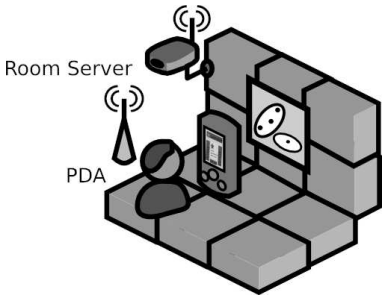
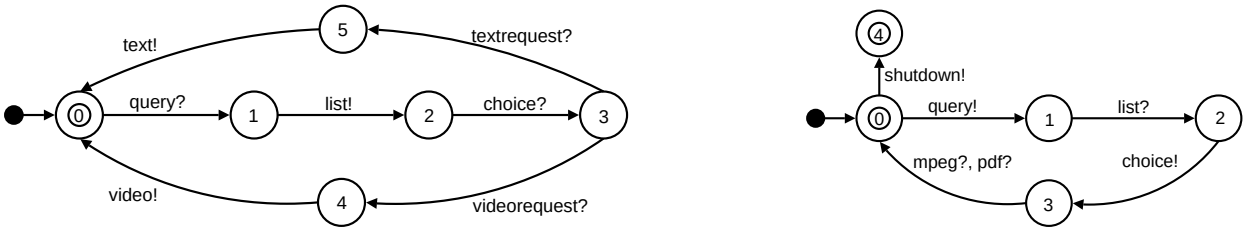


Fig. 2. The eMuseum application

Example 1: eMuseum (Fig. 2) is an added-value application whose objective is to augment the visitors' experience in museums by displaying, on their portable devices, information about seen pieces of art. We will use this example throughout the article. Let us first begin with a simplified version of it. eMuseum is built using two

separately designed components: a room server (ROOM) and a Personal Digital Assistant application (PDA). On the one hand, ROOM can be asked (query message) to send a list of artifacts present in the room (list message) and is then informed about one being selected (choice message). ROOM may afterwards be requested to send information about this artifact either in textual (pdf) or video (mpeg) format using respectively the textrequest and videorequest messages. The files themselves are sent with the text or the video message. On the other hand, PDA first issues a resource discovery query, then may be used to select a given item from a list of available resources, and the resource is eventually displayed (mpeg or pdf). PDA can be also turned off using the shutdown message.

The LTSs for these two components are given in Figure 3, with initial and final states respectively marked using bullet arrows (*e.g.*, state 0 in PDA) and hollow states (*e.g.*, states 0 and 4 in PDA). Transitions sharing the same source and target states are represented using a single transition and the list of the possible labels.



(a) ROOM LTS (b) PDA LTS

Fig. 3. eMuseum, version 1

B. Behavioural Mismatch

Mismatch situations between component interfaces may be caused by message names that do not correspond, by an ordering of messages which is not compatible in two or more components, or by some messages in one component that have no counterpart or match with several messages in another component (one-to-zero, one-to-many or many-to-one correspondences). All these cases of behavioural mismatch can be worked out using the contract notation (Section III) and the adaptation algorithms (Section IV and V) that we propose in this article. We will give examples of such mismatch in the case study we present in the sequel.

There exists numerous definitions of compatibility and, as a consequence, of mismatch between protocols [4], [19], but deadlock is the most commonly accepted notion. To automate deadlock mismatch detection, the first step is to define the semantics of a system composed of several components. This semantics can be given by means of the synchronous product [20] of LTSs. The synchronous product of several component LTSs results in a new LTS which contains all the possible interactions between the involved components, assuming they synchronise on complementary events (a,$\overline{a}$).

Definition 2 (Synchronous Product): The *synchronous product* of n LTSs $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \ldots, n\}$, is the LTS $C_1 || \ldots || C_n = (A, S, I, F, T)$ such that:

- $A = A_1 \cup \{\_\} \times \ldots \times A_n \cup \{\_\}$, $S = S_1 \times \ldots \times S_n$, $I = (I_1, \ldots, I_n)$, $F = F_1 \times \ldots \times F_n$,

- T is defined using the following rule:

  $\forall(s_1,\ldots,s_n) \in S$, $\forall i,j \in \{1,\ldots,n\}, i<j$ such that $\exists(s_i,a,s'_i) \in T_i,\ \exists(s_j,\overline{a},s'_j) \in T_j$:

  $(x_1,\ldots,x_n) \in S$ and $((s_1,\ldots,s_n),(l_1,\ldots,l_n),(x_1,\ldots,x_n)) \in T$,

  where $\forall k \in \{1,\ldots,n\}$: $\begin{cases} l_k = a, x_k = s'_i & \text{if } k = i \\ l_k = \overline{a}, x_k = s'_j & \text{if } k = j \\ l_k = \_, x_k = s_k & \text{otherwise} \end{cases}$

where the $\times$ operator stands for the cartesian product.

The states in the product correspond to sets of states of the components (called substates in the context of a product state) . For example, a state $(s_1,\ldots,s_n)$ denotes that each component C_i is its state s_i. Initially, all components are in their initial state (*i.e.*, I_i for each C_i), which means that the initial state of the product is $(I_1,\ldots,I_n)$. The computation of the transitions expresses that, given some composite state $(s_1,\ldots,s_n)$ in the product, there is some transition outgoing from this state iff there are two components, i and j, that may perform at the same time - from states s_i and s_j in their LTS - complementary events (*i.e.*, one sending a message and the other one receiving it), while other components do not perform any action (denoted _). The resulting target state of the product transition corresponds to the source state of it, but for the substates corresponding to components i and j. Transitions in the product are labelled with a set of labels, one from each component (including _). An example of synchronous product is given in Example 2, below.

We are now able to characterise mismatch by means of an adequate definition of deadlock that differentiates deadlock states and correct final states. A system is blocked when it cannot evolve and when at least one of the components is not in one of its final states.

Definition 3 (Deadlock State): Let $C = (A,S,I,F,T)$ be an LTS. A state s is a deadlock state for C, noted *dead*(s), iff it is in S, not in F and has no outgoing transitions: $s \in S \wedge s \notin F \wedge \nexists l \in A, s' \in S\ .\ (s,l,s') \in T$.

Definition 4 (Deadlock Mismatch): An LTS $C = (A,S,I,F,T)$ presents a deadlock mismatch if there is a state s in S such that *dead*(s).

To check if a system composed of several components presents mismatch, its synchronous product is computed and then Definition 4 is used. Synchronous products and deadlock detection are common in the Formal Methods community and hence are supported by tools such as CADP [21], a toolbox dedicated to the validation and verification of concurrent systems. However, our deadlock definition is slightly different from the one used in these tools, since it has to distinguish between success (deadlock in a final state), and failure (deadlock in a non-final state). Yet, behavioural mismatch detection can be automatically checked, *e.g.*, by CADP, up to the adding within component interfaces of loop transitions over final states labelled with a specific label (we use accept).

Example 2: In the synchronous product of the ROOM and the PDA components (Fig. 4), a deadlock state, (3,3), is reached after three successful interactions as this state (i) has no output transitions and (ii) is not final. The latter, (ii), is caused by the fact that the corresponding states in the ROOM (state 3) and PDA (state 3) components are not final, while both should be for (3,3) to be final. The former, (i), is caused by the name mismatch between, respectively, the PDA messages mpeg and pdf, and the ROOM messages textrequest and videorequest. One

would also note that the shutdown message in PDA has no counterpart in ROOM. Hence there is no possible sequence of transitions leading to the other potential final state in the product, *i.e.*, state (0,4), corresponding to state 0 of ROOM and state 4 of PDA.

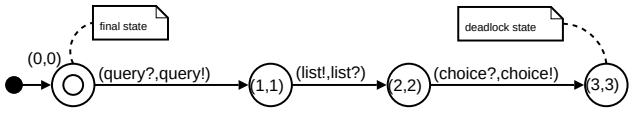


Fig. 4. Synchronous product for Example 1 LTSs

III. Adaptation Contracts

In this section, we present the adaptation contract notation that enables one to specify how to work out mismatch situations. We rely on *synchronous vectors* [20], which denote communication between several components, where each event appearing in one vector is executed by one component and the overall result corresponds to a synchronisation between all the involved components. A vector may involve any number of components and does not require interactions to occur on the same names of events. Vectors can describe expressive communication patterns, which is especially useful to express n-ary interactions.

Definition 5 (Vector): A *synchronous vector* (or *vector* for short) for a set of components $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \ldots, n\}$, is a tuple $\langle e_1, \ldots, e_n \rangle$ with $e_i \in A_i \cup \{\_\}$, _ meaning that a component does not participate in a synchronisation.

In order to identify unambiguously every communication in the adaptor, prior to the adaptation process, component event names are prefixed by the component name, *e.g.*, PDA:query!, or ROOM:query?. Yet, to favour readability, prefixes are not given in component LTS when they are clear from the context.

Example 3: Let us get back to the eMuseum example. We first define vectors for messages that match: $\mathsf{v}_{\mathrm{query}} = \langle$ROOM:query?,PDA:query!$\rangle$, $\mathsf{v}_{\mathrm{list}} = \langle$ROOM:list!,PDA:list?$\rangle$, and $\mathsf{v}_{\mathrm{choice}} = \langle$ROOM:choice?,PDA:choice!$\rangle$. Further, we have seen that mismatch came first from the unanticipated shutdown reception. This would be solved by a specific vector, $\mathsf{v}_{\mathrm{end}} = \langle$ROOM:_,PDA:shutdown!$\rangle$, to specify that the adaptor should not transmit the shutdown message to the ROOM server. Moreover, mismatch also came from the text/video choice (using textrequest or videorequest) which is not done by PDA, that waits for one resource to be sent, either with the pdf or the mpeg message. A possible solution would require to express that the video (resp. text) choice is performed by the adaptation itself using vectors $\mathsf{v}_{\mathrm{vmode}} = \langle$ROOM:videorequest?,PDA:_$\rangle$ and $\mathsf{v}_{\mathrm{tmode}} = \langle$ROOM:textrequest?,PDA:_$\rangle$. Moreover we would like to specify a correspondence between the video sending (video in ROOM) and the mpeg file reception (mpeg in PDA), and a correspondence between the text sending (text in ROOM) and the pdf file reception (pdf in PDA). The corresponding vectors would be $\mathsf{v}_{\mathrm{vget}} = \langle$ROOM:video!,PDA:mpeg?$\rangle$ and $\mathsf{v}_{\mathrm{tget}} = \langle$ROOM:text!,PDA:pdf?$\rangle$.

Vectors express correspondences between messages, like bindings between ports, or connectors, in architectural descriptions [22]. Yet, vectors alone are not sufficient to perform adaptation as one must take into account also the

context in which messages are exchanged, *i.e.*, the component protocols. Suppose we have a vector $\langle \mathsf{c}_1 : \mathsf{a}!, \mathsf{c}_2 : \mathsf{b}? \rangle$. Directly sending in an adaptor the message b to c_2 when message a is received from c_1 may lead the system to a deadlock state if this interaction is incorrect. This is why more complex adaptation algorithms, such as the ones we define in this article are required. Moreover, vectors are not sufficient to support more advanced adaptation scenarios such as contextual rules, choice between vectors or, more generally, ordering (*e.g.*, when one message in some component corresponds to several in another component, which requires to apply several vectors). The ordering in which vectors have to be applied can be specified using different notations such as regular expressions, LTSs, or (Hierarchical) Message Sequence Charts. Due to their readability and user-friendliness, we chose to specify adaptation contracts using *vector LTSs*, that is, LTSs whose labels are vectors. In addition, vector LTSs facilitate the development of adaptation algorithms since they provide an explicit description of the contract behaviours set of states, which makes their traversal easier. Other notations, such as the ones mentioned above, can be used to specify the adaptation contract, provided that they can be translated into vector LTSs. To this purpose, one can rely on existing behavioural model synthesis techniques such as those presented in [23] for regular expressions, or in [24] for Message Sequence Charts.

Definition 6 (Vector LTS): A *vector LTS* for a set of vectors V is an LTS (V, S, I, F, T) where labels are vectors.

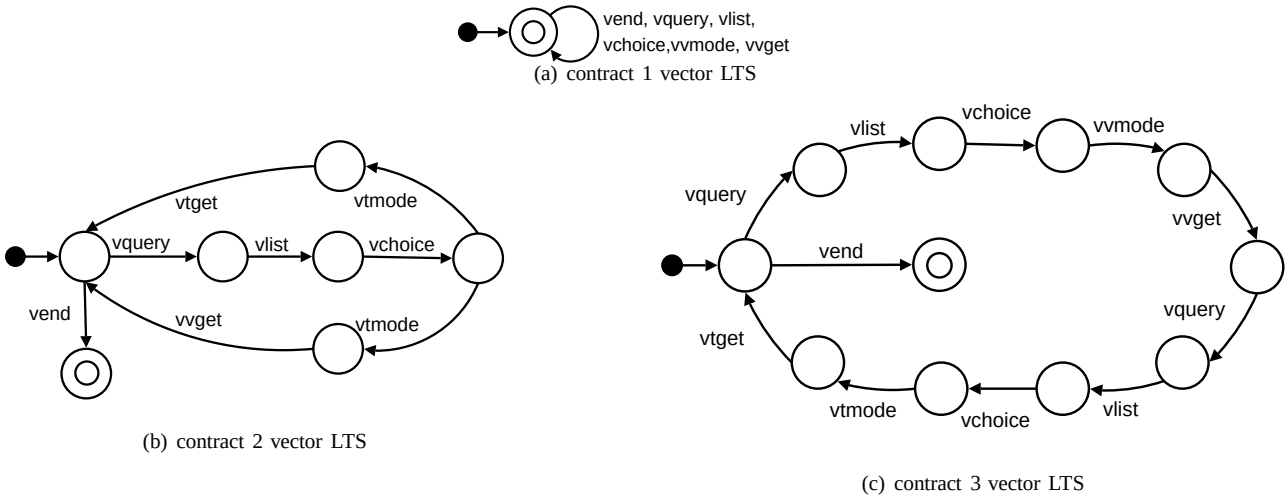
Definition 7 (Adaptation Contract): An *adaptation contract* for a set of components $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, .., n\}$, is a couple (V, L) where V is a set of vectors for components C_i, and L is a vector LTS for V.

If only message name correspondences are necessary to solve mismatch between components, the vector LTS may leave the vector application order unconstrained using a single state and all vector transitions looping on it. In particular, this pattern may be used on specific parts of the contract for which the designer does not want to impose any ordering.

The design of the adaptation contracts is the only step of adaptation which is not handled automatically by our approach. Yet, this step is essential because an inadequate contract could induce the generation of an adaptor that would ensure deadlock freedom at the cost of too many interaction removals, including ones expected by the designer. Solutions and on-going work relative to contract design are discussed in Section VIII.

Example 4: Using the vectors given in Example 3, one could express different adaptation contracts (Fig. 5). A simple example is contract 1. This contract is limited to video exchange as it does not use vectors for text exchange ($\mathsf{v}_{\mathrm{tmode}}$ and $\mathsf{v}_{\mathrm{tget}}$). But for this, the contract is very permissive. It enables any application ordering of name mismatch resolution using the vectors, including when no video is ever exchanged (*i.e.*, vectors $\mathsf{v}_{\mathrm{vmode}}$ and $\mathsf{v}_{\mathrm{vget}}$ may never be applied). One could have either text or video be exchanged with contract 2. Here, at each PDA request the adaptor will non-deterministically be able to choose between text and video. One could also enforce a very strict adaptation contract, with contract 3, where textual and video information are alternatively used. Note that the use of such highly constrained contracts, applied to adaptation without reordering, is not very interesting as giving such a contract is often close to giving the solution, while using more permissive contracts and adaptation with reordering demonstrates the full power of our automated adaptation process. Other contracts will be presented in the sequel, together with the different algorithms that operate on them to produce the corresponding adaptor

protocols.



(a) contract 1 vector LTS

(b) contract 2 vector LTS

(c) contract 3 vector LTS

Fig. 5. Adaptation contracts for eMuseum, version 1

An adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. All the exchanged messages will pass through the adaptor, which can be seen as a coordinator for the components to be adapted. This can be formalised as follows.

Definition 8 (Adaptation algorithm correctness): Given n components C_i, $i \in \{1, \ldots, n\}$ and a contract, the adaptation algorithm builds an adaptor Ad such that there is no deadlock state in the system $Ad||(C_1|| \ldots ||C_n)$.

In the sequel, we present two different correct algorithms for the generation of adaptor protocols.

IV. Adaptation without Reordering

In this section, we present a first adaptation algorithm, based on synchronous products. More precisely, we rely on an extension of the synchronous product, Def. 2, that takes into account the correspondences of events described in the vectors, but also their ordering in the vector LTS. Consequently, the vector LTS is used as a guide to build the resulting product.

Definition 9 (Synchronous Vector Product (with vector LTS)): The *synchronous vector product (with vector LTS)* of n LTS $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, .., n\}$ with a vector LTS $L = (A_L, S_L, I_L, F_L, T_L)$, is the LTS $\Pi_L((C_1, \ldots, C_n), L) = (A, S, I, F, T)$ such that:

- $A = A_L \times A_1 \cup \{\_\} \times \ldots \times A_n \cup \{\_\}$, $S = S_L \times S_1 \times \ldots \times S_n$, $I = (I_L, I_1, \ldots, I_n)$, $F = F_L \times F_1 \times \ldots \times F_n$, and
- T contains a transition $((s_L, s_1, \ldots, s_n), \langle a_L, a_1, \ldots, a_n \rangle, (s'_L, s'_1, \ldots, s'_n))$ iff there is a state $(s_L, s_1, \ldots, s_n)$ in S, there is a transition $(s_L, \langle l_1, \ldots, l_n \rangle, s'_L)$ in T_L and for every i in $\{1, \ldots, n\}$:
  - if $l_i = \_$ then $s'_i = s_i$ and $a_i = \_$,
  - otherwise there is a transition (s_i, a_i, s'_i) with $a_i = l_i$ in T_i.

Algorithm 1 *build_adaptor_noreordo*

constructs an adaptor without reordering for a set of components given an adaptation contract

inputs components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, and an adaptation contract (V, L)

output adaptor $Ad = \langle A, S, I, F, T \rangle$

1: $P_L := \Pi_L((C_1, \ldots, C_n), L)$ *// product with vector LTS L*

2: $P =$*proj*(P_L) *// discarding vector LTS elements in the result*

3: $P_{restr} :=$ *remove_deadlocks*(P) *// removing paths leading to deadlocks*

4: $S_{\text{add}} := \emptyset$

5: $T_{Ad} := \emptyset$

6: **for all** $t = (s = (s_1, \ldots, s_n), (l_1, \ldots, l_n), s' = (s'_1, \ldots, s'_n))$ in $T_{P_{\text{restr}}}$ **do**

7: $L_{\text{rec}} = \{l? \mid l! \in (l_1, \ldots, l_n)\}$ *// mirroring: emissions to receptions*

8: $L_{\text{em}} = \{l! \mid l? \in (l_1, \ldots, l_n)\}$ *// mirroring: receptions to emissions*

9: $\text{Seq}_{\text{rec}} =$ *compute_permutations*(L_{rec}) *// permutations between receptions*

10: $\text{Seq}_{\text{em}} =$ *compute_permutations*(L_{em}) *// permutations between emissions*

11: **for all** $(R = (r_1, \ldots, r_i), E = (e_1, \ldots, e_p)) \in \text{Seq}_{\text{rec}} \times \text{Seq}_{\text{em}}$ **do**

12: $T_{Ad} := T_{Ad} \cup \{s \xrightarrow{r_1} q_1, \ldots, q_{i-1} \xrightarrow{r_i} q_i, \ldots, q_{i+1} \xrightarrow{e_1} q_{i+2}, \ldots, q_{n-1} \xrightarrow{e_p} s'\}$

13: $S_{\text{add}} := S_{\text{add}} \cup \{q_1, \ldots, q_{n-1}\}$

14: **end for**

15: **end for**

16: **return** $Ad = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}} \cup S_{\text{add}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_{Ad})$

As with Def. 2, states in the product correspond to sets of states of the components, but take also into account the vector LTS. For example, a state $(s_0, s_1, \ldots, s_n)$ denotes that each component C_i is in its state s_i and that the vector LTS is in s_0. Initially all components and the vector LTS are in their initial state (*i.e.*, I_i for each C_i and I_L for the vector LTS), which means that the initial state of the product is $(I_L, I_1, \ldots, I_n)$. The computation of the transitions is also slightly different from Def. 2. There is an outgoing transition from a state $(s_L, s_1, \ldots, s_n)$ iff there is a transition labelled by a vector $\langle l_1, \ldots, l_n \rangle$ outgoing from state s_L in the vector LTS and, as a consequence, if for every component C_i there is a transition outgoing from s_i and labelled with l_i in the C_i LTS. A commented example of synchronous vector product computation is given in Example 5, Figure 8.

To generate an adaptor protocol from a synchronous vector product we have to discard the first element of the product components to keep only the elements corresponding to the component exchanges. More formally, it means that from an LTS $P_L = \Pi_L((C_1, \ldots, C_n), L) = (A, S, I, F, T)$ we compute the LTS $P =$ *proj*$(P_L) = (A', S', I', F', T')$ such that $\forall X \in \{A, S, I, F\}$ $X' = \{$*cdr*$(x) \mid x \in X\}$ and $T' = \{($*cdr*(s),*cdr*(l),*cdr*$(s')) \mid (s, l, s') \in T\}$ with *cdr*$((x_0, x_1, \ldots, x_n)) = (x_1, \ldots, x_n)$.

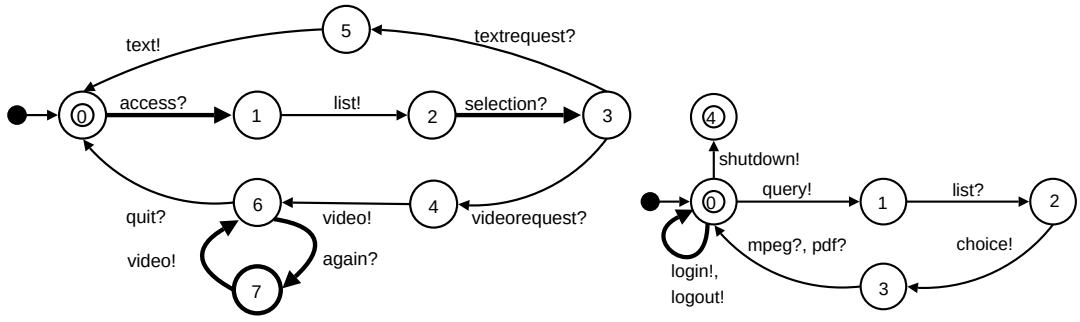
Our algorithm (Alg. 1) takes as input a set of component LTSs C_i and an adaptation contract (V, L). This algorithm is based on three main steps: (i) computation of the synchronous vector product taking the vector LTS L into account, and discarding in the result the vector LTS elements (Alg. 1:1–2), (ii) removal of interaction sequences (paths) leading to deadlock (function *remove_deadlocks*, Alg. 1:3), and (iii) for each transition (Alg. 1:6–15), reversal of the directions for all events appearing in the vector on the transition, called mirroring (Alg. 1:7–8), and computation of all possible interleavings (function *compute_permutations*) starting with receptions (Alg. 1:9–14).

Removing deadlock paths is required to suppress spurious interactions that would not leave the system in a stable (final) state, as shown in Example 5 below. This is achieved recursively removing transitions and states yielding deadlocks: find a state s such that *dead*(s), remove s and any transition t with target s, and do this until there is no more such s in the LTS. Mirroring ensures that the adaptor and the components can perfectly communicate using the same event message names with opposite directions (!/? or ?/!). Moreover, event interleaving is essential when vectors involve more than two events in a communication (*e.g.*, in case of broadcast or multicast communication). Interleavings make the adaptor support non-determinism *wrt.* the orderings in which events will occur, hence accept any possible one.

Note that Algorithm 1 builds an adaptor protocol by applying one vector after the other, that is, all interactions involved in one vector occur before starting the interactions of another vector. Consequently, events belonging to two vectors appearing as labels in the synchronous product are never interleaved. Such an interleaving is mandatory when events need to be reordered. This additional feature will be supported by the algorithm presented in Section V. The complexity of Algorithm 1 lies on the synchronous vector product computation, and is $\mathbf{O}(|S|^{n+1})$ where S is the largest set of states for all component (and vector) LTS, and $n+1$ stands for the n components plus the vector LTS. The proof of correctness of Algorithm 1 can be found in Appendix II.

Example 5: Let us now present a second version of eMuseum. A new version of the ROOM component supports an additional feature: once a video has been sent, it can be re-sent (upon reception of the again message) to be played again. The quit message is then used to tell ROOM one is done with the selected video. The ROOM designer has also refactored this component. The names of some operations (namely, query and choice) and, as a consequence, of the corresponding messages, have been changed. A new version of the PDA component is also used. It now supports to be integrated in contexts where rights can be different depending on two modes: a guest mode (with less rights) and a user mode (with more rights). PDA can send login (respectively logout) messages to go from guest to user mode (respectively from user to guest mode). The new interfaces of the two components are given in Figure 6 (changes are in bold).

As far as the adaptation contract is concerned, one does not start from scratch. The vectors we had before are reused, replacing old messages by new ones where we have now name mismatch (in bold font): $\mathsf{v}_{\mathrm{end}} = \langle$ROOM:_,PDA:shutdown!$\rangle$, $\mathsf{v}_{\mathrm{vmode}} = \langle$ROOM:videorequest?,PDA:_$\rangle$, $\mathsf{v}_{\mathrm{vget}} = \langle$ROOM:video!,PDA:mpeg?$\rangle$, $\mathsf{v}_{\mathrm{tmode}} = \langle$ROOM:textrequest?,PDA:_$\rangle$, $\mathsf{v}_{\mathrm{tget}} = \langle$ROOM:text!,PDA:pdf?$\rangle$, $\mathsf{v}_{\mathrm{query}} = \langle$**ROOM:access?**,PDA:query!$\rangle$, $\mathsf{v}_{\mathrm{list}} = \langle$ROOM:list!, PDA:list?$\rangle$, and $\mathsf{v}_{\mathrm{choice}} = \langle$**ROOM:selection?**,PDA:choice!$\rangle$.



(a) ROOM LTS (b) PDA LTS

Fig. 6. eMuseum, version 2

We also add vectors for unspecified receptions of messages sent by the PDA for changing mode, as ROOM has not been built *wrt.* such modes: $\mathbf{v_{user}} = \langle$**ROOM:_,PDA:login!**$\rangle$ and $\mathbf{v_{guest}} = \langle$**ROOM:_,PDA:logout!**$\rangle$. The support for changing mode, and more generally contexts will be achieved using the vector LTS, below. Finally, we add vectors corresponding to the new feature of ROOM (re-sending videos): $\mathbf{v_{again}} = \langle$**ROOM:again?,PDA:_**$\rangle$ and $\mathbf{v_{quit}} = \langle$**ROOM:quit?,PDA:_**$\rangle$. The adaptor will be in charge of sending them when required, as for the video and text requests. Note that if we had used a single vector $\langle$ROOM:quit?,PDA:shutdown!$\rangle$ in place of $\mathrm{v_{end}}$ and $\mathrm{v_{again}}$, we would have enforced that ROOM and PDA exchange information exactly once (forbidding the PDA to shut down directly and to ask several times information).

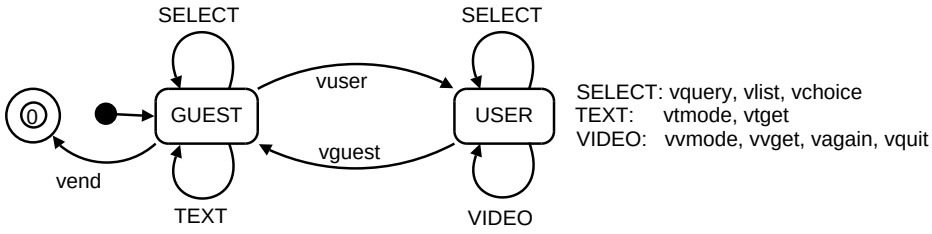


Fig. 7. Adaptation contract for eMuseum, version 2

By using a vector LTS (Fig. 7), we will enforce the following constraints:

- there are two modes, GUEST and USER. In the eMuseum application, we take benefit of these two modes as follows. In GUEST mode the sent information is text. In USER mode, the sent information is video. This demonstrates how an adaptation contract can be used to enforce constraints which are defined system-wide, not at the level of individual components;
- the two modes alternate (starting in GUEST mode), with going from one to another using the login and logout messages;
- we know that communication is based on two phases, selection and getting information, yet we keep an abstract description level for these. Non-determinism may be kept in the contract, *e.g.*, in USER mode, between different

possible application orderings of v_{query}, v_{list}, v_{choice}, v_{vmode}, v_{vget}, v_{again}, and v_{vquit} to let the adaptation process decide which one – if any – is correct (see the corresponding adaptor, Fig. 9). For this, the adaptation process uses the orderings which are defined in the component interfaces.

In order to generate the adaptor protocol, we first compute the synchronous vector product (Fig. 8) of the ROOM LTS (Fig. 6(a)) and the PDA LTS (Fig. 6(b)) with the vector LTS (Fig. 7). To understand how this works, let us take for example the computation of the transitions outgoing from the product initial state. This initial state, (0,0,GUEST), corresponds to the composition of the components' and vector's LTS initial states. Different sets of transitions are possible in the three LTSs used in the product:

- access? in ROOM;
- shutdown!, login!, logout!, and query! in PDA;
- vectors v_{end} (⟨ROOM:_, PDA:shutdown!⟩), v_{user} (⟨ROOM:_, PDA:login!⟩), v_{query} (⟨ROOM:access?, PDA:query!⟩), v_{list} (⟨ROOM:list!, PDA:list?⟩), v_{choice} (⟨ROOM:selection?, PDA:choice!⟩), v_{tmode} (⟨ROOM:text-request?, PDA:_⟩), and v_{tget} (⟨ROOM:text!, PDA:pdf?⟩) in the vector LTS.

Therefore, there are only three possible transitions outgoing from the product initial state (corresponding to the first three vectors above):

- ⟨ ⟨ROOM:_,PDA:shutdown!⟩, ROOM:_, PDA:shutdown! ⟩, going to state (0,4,0);
- ⟨ ⟨ROOM:_,PDA:login!⟩, ROOM:_, PDA:login! ⟩, going to state (0,0,USER);
- ⟨ ⟨ROOM:access?,PDA:query!⟩, ROOM:access?, PDA:query! ⟩, going to state (1,1,GUEST).

The other possibilities are forbidden, either because one component corresponding to a message in a possible vector is not ready for it (*e.g.*, ROOM cannot receive textrequest in its initial state, 0) or because components may be ready for some message but the contract forbids it (*e.g.*, PDA may send logout but vector v_{guest} is not enabled in the initial state of the vector LTS, (0,0,GUEST)). We may proceed similarily, step by step, computing for example now the transitions outgoing from the (0,4,0), (0,0,USER), and (1,1,GUEST) states. The result is given in Figure 8 where the part of the labels corresponding to the vectors are discarded due to place matters (*i.e.*, *wrt.* Alg. 1, we give P in place of P_L).

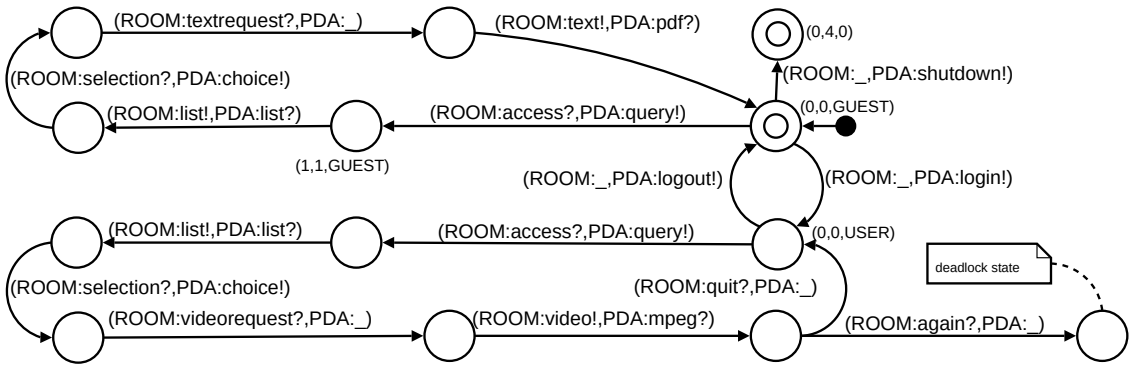


Fig. 8. Product LTS for eMuseum, version 2

One may note a path leading to a deadlock state on this example. After ROOM and PDA have successfully exchanged a first video, the adaptor may have ROOM send it again using the again message. However, ROOM would then send the video message which would block the system as PDA is not ready to receive the corresponding mpeg message. Indeed this could have been prevented by removing vector v_{again} from the adaptation contract. Yet, as one cannot ensure the perfect contract is always given, it shows the need for the suppression of spurious interactions after the product is computed.

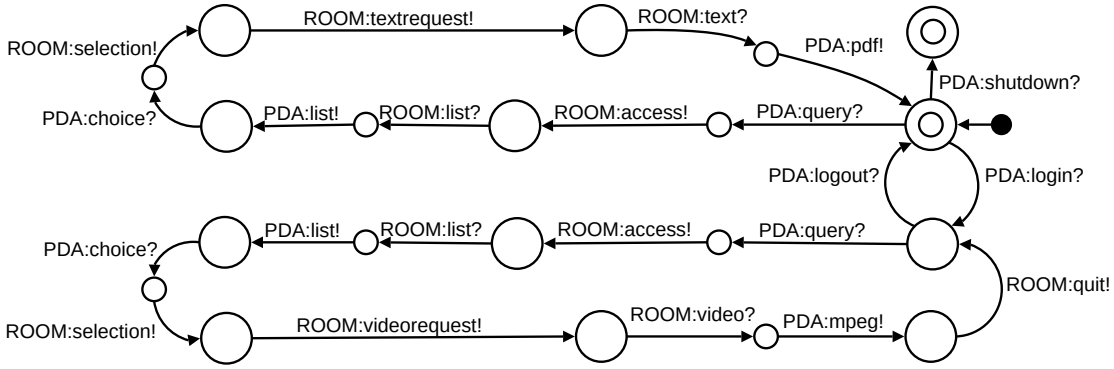


Fig. 9. Adaptor protocol for eMuseum, version 2

We finally compute the adaptor by mirroring the labels and computing permutations of inputs and then outputs for it, see Figure 9. In this adaptor protocol, we see that behavioural mismatch (one-to-zero, *i.e.*, unanticipated reception, such as shutdown, one-to-one such as query *vs* access, many-to-one such as videorequest and video *vs* mpeg) have been worked out. The adaptor follows a coordination protocol which is restricted to the contract and modes we specified (*e.g.*, text is sent to guests and video to users). Finally, the adaptor has also removed all possible interaction sequences leading to deadlocks (as demonstrated above with video re-sending).

V. Adaptation with Reordering

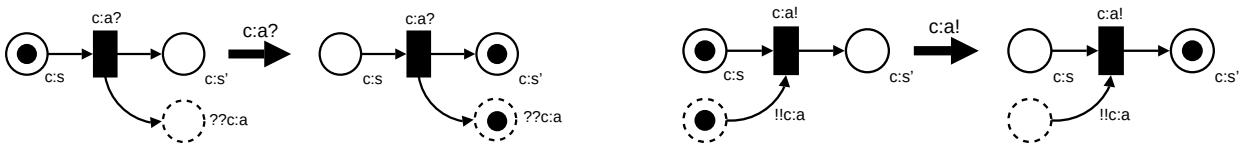
Let us now extend the domain of mismatch problems we deal with. Our goal is now also to address behavioural mismatch which requires reordering. This occurs when exchanged messages present non-compatible orderings in the components' protocols. To support this kind of mismatch, the adaptation process may try to accommodate protocols by reordering events in-between the components. The behavioural adaptation proposal presented in Section IV may yield an empty adaptor in presence of such mismatch because it induces application of one vector after the other, and therefore prevents the application of several vectors at the same time that is necessary to make reordering effective.

To this purpose, we present a second approach which complements the one presented in Section IV. Messages received by the adaptor are seen as *resources* which are memorised until they need to be sent (*i.e.*, until they may be received by some component to make it evolve). This can be achieved first thanks to an *encoding* of the component protocols and of the adaptation contract into a formalism that supports a *memory* and a *resource-based*

vision of adaptation, as follows: (i) reception of messages (by the adaptor) corresponds to a resource creation, (ii) emission of messages (by the adaptor) is possible provided some resource is available and corresponds to resource consumption, and finally, (iii) vectors correspond to resource transfer. Petri nets [25] are such a formalism, which further benefits from good tool support. Moreover, the marking graph of such a Petri net encoding represents all possible resource-based evolutions of the adaptor (message reception, emission and transfer).

Before presenting our algorithm for adaptation with reordering in more details, let us introduce first the basics of Petri nets. A Petri net consists of places, transitions and directed arcs between places and transitions. A transition is connected by input arcs to a set of input places, and by output arcs to a set of output places. Places may contain any number of tokens that model resources. Transitions act on tokens by a process known as *firing*. A transition can be fired if there are enough tokens in each of its input places. When a transition fires, it consumes one token from each of its input places, and adds a token into each of its output places. The presentation of Petri nets is simplified here for conciseness purposes as, *e.g.*, generalised Petri nets support arcs labelled with natural numbers to denote the need of more than one token in an input place and the production of more than one token in an output place. A distribution of tokens over the places of a net is called a *marking*. A *marking graph* describes all the markings that can be reached from an initial marking by firing transitions.

Algorithm 2 takes as input a set of component LTSs C_i and an adaptation contract, and generates the corresponding Petri net encoding. As regards component interface encoding (Fig. 10, Alg. 2:2–12), every event emission or reception in a component is translated into a Petri net transition holding the same name as the event but the reversed direction. This transition is connected to specific places that are used to store, using tokens, messages corresponding to the events. For each event emission $c:a!$ in a component c interface (Fig. 10(a)), there is a transition for reception in the Petri net ($c:a?$) and this transition has an output arc to the place where the corresponding message is stored ($??c:a$). Conversely, for each event reception $c:a?$ in a component c interface (Fig. 10(b)), there is a transition for emission in the Petri net ($c:a!$) and this transition has an input arc from the place where the corresponding message has been stored ($!!c:a$). The control flow between events in component interfaces is expressed in the Petri net by control places and related arcs connecting the different Petri net transitions. Moreover, tokens are placed in the control places encoding the initial states of the LTS interfaces (Alg. 2:4), and their evolution will simulate the execution of the entire system.



(a) Message emission in components / reception in adaptor

(b) Message reception in components / emission in adaptor

Fig. 10. Encoding patterns for component protocols (and related marking evolution semantics)

As far as the contract encoding is concerned (Alg. 2:13–24), every synchronous vector is encoded using a tau [18]

Algorithm 2 *build_PetriNet*

constructs a Petri net encoding from component interfaces and an adaptation contract

inputs components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, and an adaptation contract $(V, L = (A_L, S_L, I_L, F_L, T_L))$

output Petri net $\mathcal{N}$

1: $\mathcal{N} :=$ *empty_PetriNet*() *// all the following actions operate on* $\mathcal{N}$
2: **for all** $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle, i \in \{1, \ldots, n\}$ **do**
3: **for all** $s_j \in S_i$ **do** add a place $c_i : s_j$ **endfor**
4: put a token in place $c_i : I_i$ *// I_i is the initial state of C_i*
5: **for all** $a! \in A_i$ **do** add a place $??c_i : a$ **endfor**
6: **for all** $a? \in A_i$ **do** add a place $!!c_i : a$ **endfor**
7: **for all** $(s, e, s') \in T_i$ with $l = e$ **do**
8: add a transition with label $\overline{l}$, an arc from place $c_i : s$ to the transition and an arc from the transition to place $c_i : s'$
9: **if** l has the form $a!$ **then** add an arc from the transition to place $??c_i : a$ **endif**
10: **if** l has the form $a?$ **then** add an arc from place $!!c_i : a$ to the transition **endif**
11: **end for**
12: **end for**
13: **for all** $s_L \in S_L$ **do** add a place $c_L : s_L$ **endfor**
14: put a token in place $c_L : I_L$ *// I_L is the initial state of L*
15: **for all** $t_L = (s_L, \langle e_1, \ldots, e_n \rangle, s'_L) \in T_L$ with $\forall i \in \{1, \ldots, n\}\ l_i = e_i$ **do**
16: add a transition with label tau, an arc from place $c_L : s_L$ to the transition and an arc from the transition to place $c_L : s'_L$
17: **for all** l_i **do**
18: **if** l_i has the form $a!$ **then** add an arc from place $??c_i : a$ to the transition **endif**
19: **if** l_i has the form $a?$ **then** add an arc from the transition to place $!!c_i : a$ **endif**
20: **end for**
21: **end for**
22: **for all** $(f_r, f_1, \ldots, f_n) \in F_L \times F_1 \times \ldots \times F_n$ **do**
23: add a (loop) accept transition with arcs from and to each of the tuple elements
24: **end for**
25: **return** $\mathcal{N}$

transition (Fig. 11, Alg. 2:16–20) as it represents an internal action of the adaptor. Arcs are added (Alg. 2:16) to connect these tau transitions in order to enforce their application ordering in the vector LTS. Message transfer is

Algorithm 3 *build_adaptor_reordo*

constructs an adaptor with reordering for a set of components given an adaptation contract

inputs components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, and an adaptation contract (V, L)

output adaptor $Ad = \langle A, S, I, F, T \rangle$

1: $\mathcal{N} :=$ *build_PetriNet*$(\{C_1, \ldots, C_n\}, (V, L))$ *// see Algorithm 2*

2: $M :=$ *get_marking_graph*$(\mathcal{N})$

3: $Ad :=$*reduction*(*remove_deadlocks*(M))

4: **return** Ad

enabled using input/output arcs that connect a tau transition to the places related to the component events involved in the corresponding vector (Alg. 2:17–20).

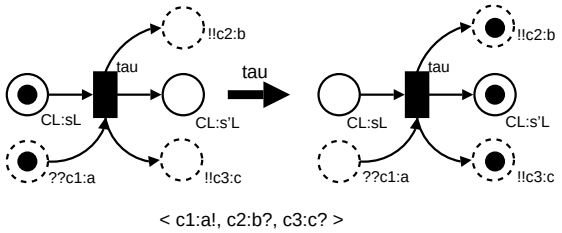


Fig. 11. Encoding pattern for adaptation contracts (and related marking evolution semantics)

We will illustrate further in this section (Ex. 6) this encoding into Petri nets on the eMuseum application.

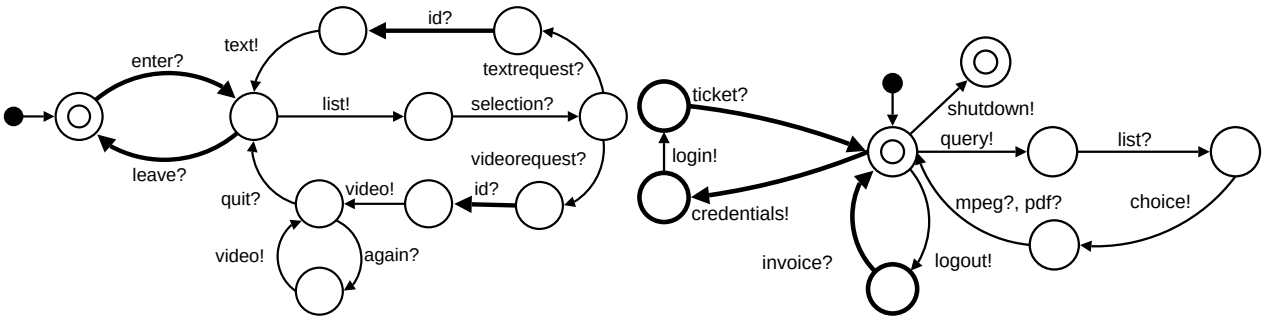
Algorithm 3 generates an adaptor protocol from a set of component LTSs C_i and an adaptation contract. This algorithm respectively (i) builds a Petri net encoding for both component LTSs and the contract (Alg. 3:1), (ii) generates the marking graph for this Petri net which contains all the possible evolutions of the adaptor *wrt.* the component LTSs it is in charge of (Alg. 3:2), and (iii) removes remaining deadlocks (*remove_deadlocks*) which correspond to spurious interactions, and tau transitions (*reduction*) introduced during the Petri net generation (Alg. 3:3).

The *reduction* function is used to simplify the adaptor protocols. At this level, several behavioural reductions modulo an equivalence relation can be applied (*e.g.*, tau*.a, observational, branching). In our experiments, we used in particular a combination of branching and weak trace reductions that enable (i) to eliminate tau transitions introduced for message transfer in the encoding of vectors into Petri nets (which are meaningless at the level of the adaptor) while preserving the deadlock freedom property, (ii) to cut similar paths (traces), and (iii) to determinize the adaptor protocols using a classical automata theory algorithm.

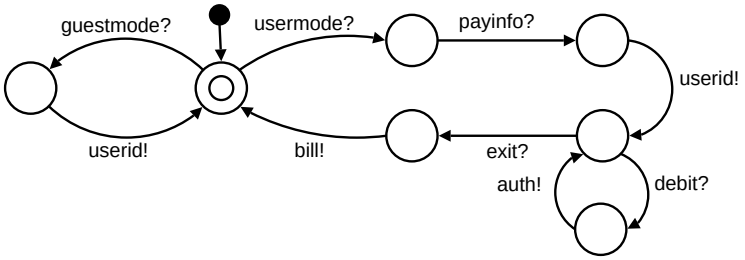
The theoretical complexity of this algorithm lies mainly in the marking graph construction, which is exponential [26]. In practice, it is less expensive as parts of the net are 1-bounded (there is only one token in only one of the places corresponding to the component interface states). We emphasise that the adaptation techniques presented

in this section work also for adaptation without reordering. However, since the computational complexity of these techniques is greater than those presented in the former section, they are privileged only if reordering is needed. The proof of correctness of Algorithm 3 can be found in Appendix II.

Example 6: Let us now describe the last version of the eMuseum application. A third component, a generic pay-per-view subscription server, SUB, is used to manage subscription modes (guest mode for free access and user mode for paying access) and related access identifiers. Upon reception of a registration message (guestmode or usermode), it returns an access identifier (userid message). In case of user registration, reception of the payment information (payinfo message) is required before sending the identifier. Moreover, using debit, the user shopping cart can be updated (with an access authorization sent back each time) before a bill is finally sent (the user account being debited at the same time). There are also changes in new versions of the other two components which are reused. ROOM needs an identifier (id) to be given before information sending in order to update a log file. The access to ROOM is controlled by a signal detecting the entry (enter) and the leaving (leave) of the room. PDA sends payment information (credentials) before logging in and waiting for an acknowledgement (ticket). Finally, after logging out, PDA waits for an invoice of the services it acceded to.



(a) ROOM LTS (b) PDA LTS



(c) SUB LTS

Fig. 12. eMuseum, version 3

The new corresponding LTSs are given in Figure 12 (changes are in bold). One may note that:

- PDA does not deal with identifiers when doing requests, while ROOM needs them (id?);
- ROOM knows nothing about guest and user modes;
- reordering is required, first because PDA and ROOM do not support requests in the same way: PDA sends a

query before waiting a list of items and selecting one, while ROOM presents its list of items and waits for one to be selected before waiting for either a text or a video request. Moreover, PDA and SUB do not treat the logging in protocol in the same way, the order of the payment information and login request being different in both components.

To work this out, vectors are first defined (differences with the previous example are in bold font). There are two new vectors for the entering and leaving of the system (triggered by the adaptor), while the one for PDA shutdown is reused. In the first case, entering also triggers the guest mode (initial mode).

$\mathbf{v_{enter}}$= ⟨**ROOM:enter?,PDA:_,SUB:guestmode?**⟩, $\mathbf{v_{leave}}$= ⟨**ROOM:leave?,PDA:_,SUB:_**⟩, and

v_{end} = ⟨ROOM:_,PDA:shutdown!,SUB:_⟩.

Vectors for lists and choices are also reused:

v_{list} = ⟨ROOM:list!,PDA:list?,SUB:_⟩ and v_{choice} = ⟨ROOM:selection?,PDA:choice!,SUB:_⟩.

Vectors for entering text (resp. video) mode and for text (resp. video) exchange are reused but for two differences: (i) query in PDA now corresponds to requests in ROOM, and (ii) SUB should be informed about each video being sent:

v_{tmode} = ⟨ROOM:textrequest?,**PDA:query!**,SUB:_⟩, v_{tget} = ⟨ROOM:text!,PDA:pdf?,SUB:_⟩,

v_{vmode} = ⟨ROOM:videorequest?,**PDA:query!**,**SUB:debit?**⟩, v_{vget} = ⟨ROOM:video!,PDA:mpeg?,**SUB:auth!**⟩, and

v_{quit} = ⟨ROOM:quit?,PDA:_,SUB:_⟩.

Vectors for changing mode are reused and modified to support SUB:

v_{user} = ⟨ROOM:_,PDA:login!,**SUB:usermode?**⟩ and v_{guest} = ⟨ROOM:_,PDA:logout!,**SUB:guestmode?**⟩.

Vectors that support the additional payment relations between PDA and SUB are added:

$\mathbf{v_{info}}$= ⟨**ROOM:_,PDA:credentials!,SUB:payinfo?**⟩, $\mathbf{v_{bill}}$= ⟨**ROOM:_,PDA:invoice?,SUB:bill!**⟩, and

$\mathbf{v_{exit}}$= ⟨**ROOM:_,PDA:_,SUB:exit?**⟩.

Identifier exchange is finally specified with three vectors (one for guest mode, one for user mode and one for re-sending):

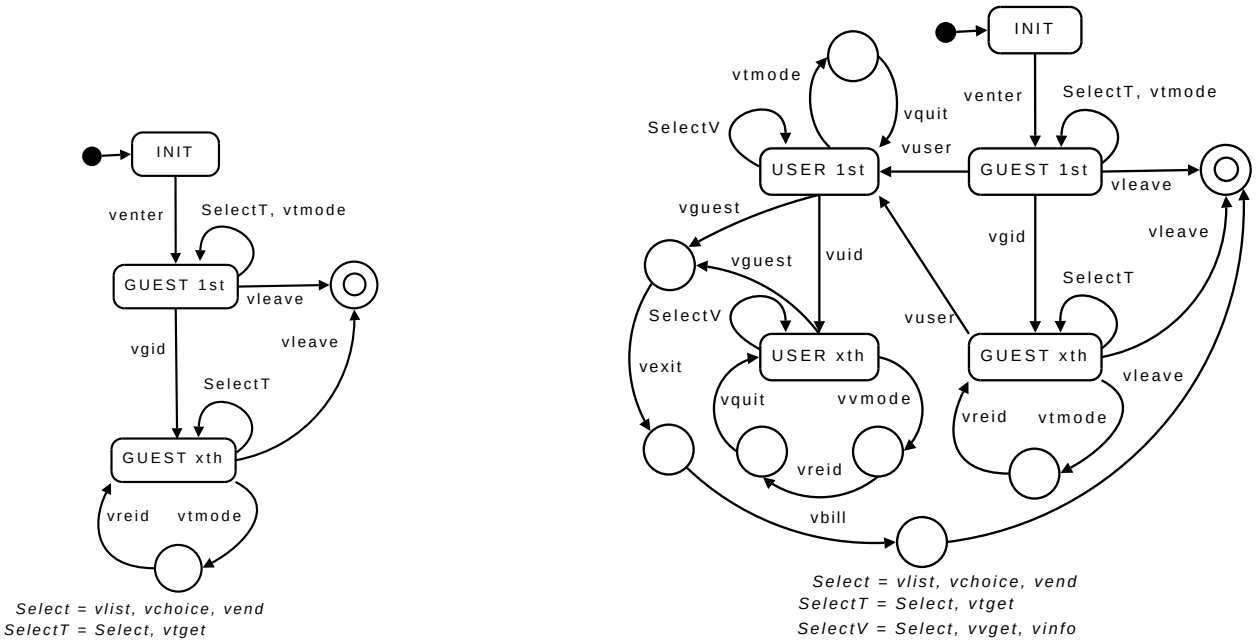
$\mathbf{v_{gid}}$= ⟨**ROOM:id?,PDA:_,SUB:userid!**⟩, $\mathbf{v_{uid}}$= ⟨**ROOM:id?,PDA:ticket?,SUB:userid!**⟩, and

$\mathbf{v_{reid}}$= ⟨**ROOM:id?,PDA:_,SUB:_**⟩.

Vector v_{again} is left over, suppressing the possibility for video re-sending.

As for the previous example, we may now use a vector LTS to specify their possible orderings. We propose two different contracts: one supporting only the GUEST mode (Fig. 13(a)) and one supporting both modes (Fig. 13(b)).

The contract for the GUEST mode (Fig. 13(a)) focuses on what happens between one enters and one leaves the room. Moreover, it specifies that once the identifier has been first exchanged, the identifier is re-sent by the adapter (vector $v_{\mathbf{reid}}$) only if a new query happens (vector $v_{\mathbf{tmode}}$). But for these two constraints, the contract is not restrictive and does not specify any particular ordering of vectors. The adaptation process will therefore find all possible ones such that the adapted system does not deadlock. The contract for the full mode (Fig. 13(b)) adds a part relative to the USER mode. One may note that it is symmetric to the GUEST mode contract but for



(a) GUEST mode vector LTS

(b) full mode vector LTS

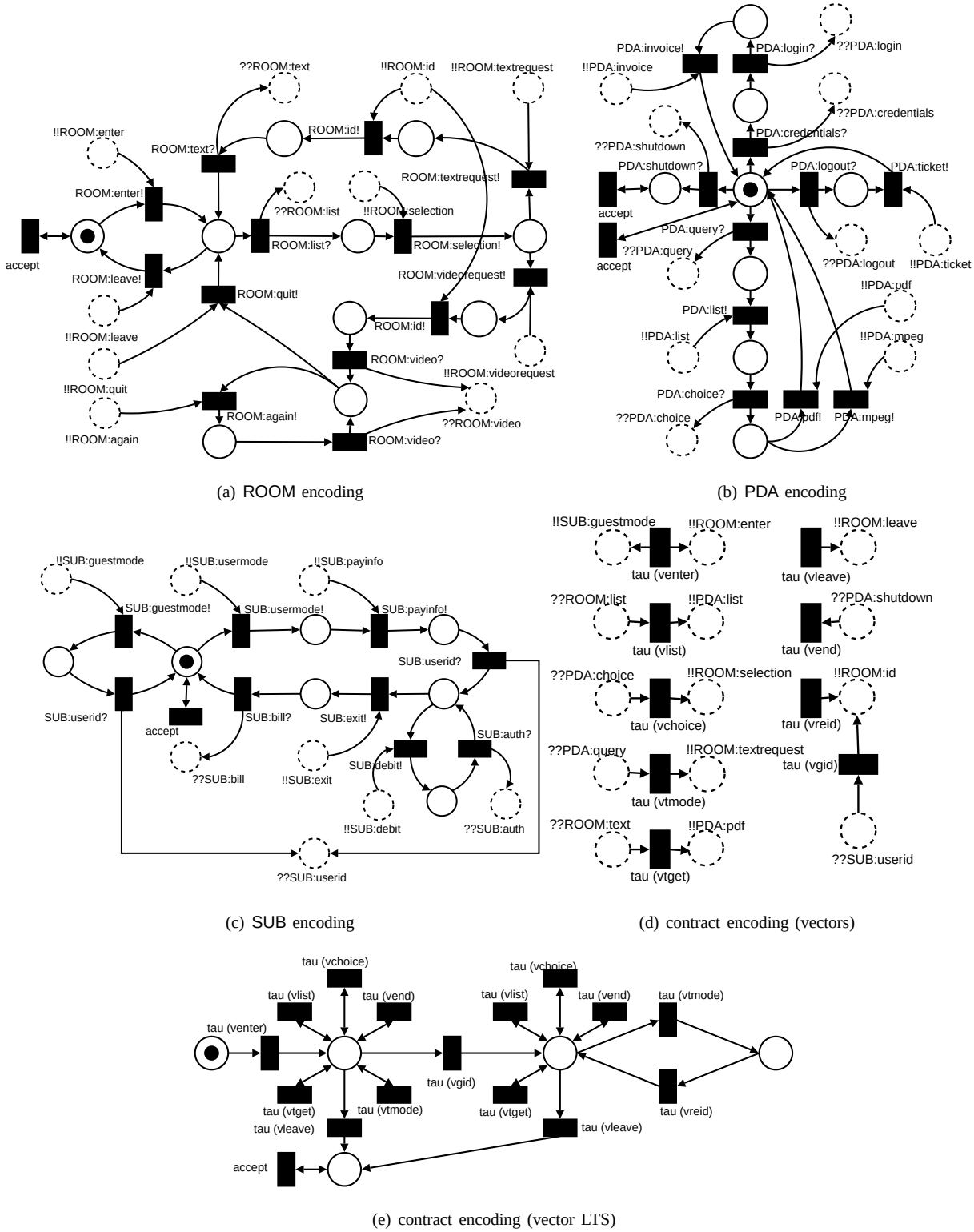
Fig. 13. Adaptation contracts for eMuseum, version 3

some differences. We must first take into account the quit message emission by the adaptor (vector $\mathbf{v_{quit}}$) to avoid blocking once a video has been exchanged. This is put into practice, *e.g.*, by adding this vector in the vector LTS at the end of the $\mathbf{v_{tmode}}$ loops (twice). Moreover, while passing from GUEST to USER mode is quite simple (vector $\mathbf{v_{user}}$), leaving USER mode should also take into account the final payment using vectors $\mathbf{v_{bill}}$ and $\mathbf{v_{exit}}$. This is representative of one-to-many correspondence, here between logout in PDA and both exit and guestmode in SUB. The obtaining of the full mode contract (and the difference between the USER and the GUEST modes) has been achieved in several steps, using post-generation adaptor assessment (see support for contract design in Section VIII). In the sequel, we will present our approach on the first contract due to the complexity of the adaptor for the full mode.

The Petri net generated for this example is given in Figure 14. To help the reader, we present separately the parts of the Petri net which are generated for ROOM, PDA, SUB, and the contract. The nets are glued on dashed places, accept transitions and, for the contract, on vector transitions.

The adaptor for the GUEST mode has 204 states and 404 transitions (494 states and 1101 transitions before pruning paths to deadlocks). After reduction, the resulting final adaptor has 52 states and 104 transitions (Fig. 15, where the initial state is in light gray and the final states are in black). We emphasize that it is much simpler to give an adaptation contract and use our automatic adaptor protocol generation techniques than writing directly the protocol by hand.

One may note different things (see the zoom in Fig. 15):



(a) ROOM encoding

(b) PDA encoding

(c) SUB encoding

(d) contract encoding (vectors)

(e) contract encoding (vector LTS)

Fig. 14. Petri net encoding for eMuseum, version 3 (GUEST mode)

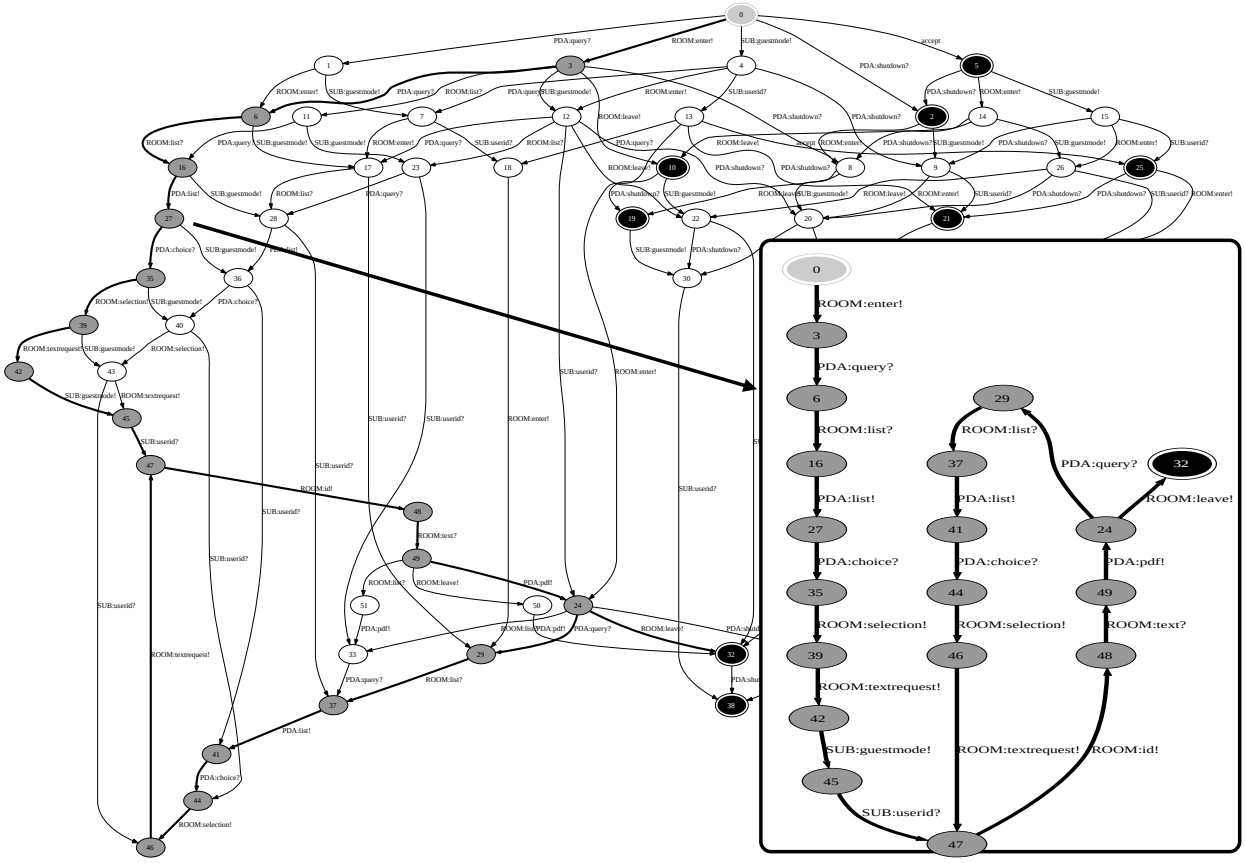


Fig. 15. Adaptor protocol for eMuseum, version 3 (GUEST mode)

- name mismatch is solved, *e.g.*, choice in PDA *vs* selection in ROOM;
- messages are reordered when required, *e.g.*, PDA sending query and then waiting for the list of possible information to be displayed, while ROOM sending first the list and waiting for a selection before waiting for either a textrequest or videorequest which correspond to query;
- id is re-sent to ROOM when required;
- the contract is permissive – *e.g. wrt.* in which order to apply vectors v_{list}, v_{choice}, v_{tmode}, v_{tget} and v_{vend} – and the adaptor contains all possible orderings not leading to deadlocks (yet we only have represented one possible ordering on the zoom).

The adaptor for the full mode has 1477 states and 3326 transitions (2719 states and 6464 transitions before pruning paths to deadlocks). After reduction, the resulting final adaptor has 307 states and 627 transitions. Due to its size, the adaptor is given in Appendix I. Performing verification on the adapted system (made up of the components and the adaptor) we have been able to check with CADP that important system-level properties are enforced through adaptation: (i) no video is sent before the PDA logs on, and (ii) a debit is performed for each video being sent.

VI. The Adaptor Tool

The approach for software adaptation that we have presented in the former sections of this article has been implemented in a tool, called Adaptor [27]. The kernel of Adaptor corresponds to the implementation of the algorithms that generate adaptor protocols being given behavioural interfaces of components and an adaptation contract. In addition, Adaptor presents graphical interfaces to load and visualise the different inputs, to apply the different adaptation steps, and to visualise the intermediate encodings and final results. The tool was initially developed in Python (about 9,000 lines of code[1]), and uses GTK+ technology for the development of the user interface. More recently, to simplify the access and use of the tool, a Web service version of Adaptor (WS-Adaptor) has been implemented in Java. It enables one to adapt component protocols without installing more than a GUI client (the engine and the required dependencies running in the distant Web service host).

Different input and output formats are used to describe respectively interfaces of components, contracts, and resulting adaptors. As regards inputs, LTS interfaces may be described using XML or the Aldebaran textual format [21] (file extension .aut). Vectors and vector LTSs involved in contracts are specified using XML.

Once the inputs are loaded, Adaptor uses dot [28] (graphviz) to visualise interfaces of components, intermediate results for contracts, Petri nets, and adaptors. Textual formats are also possible for visualisation, or storing and analysis purposes, namely .aut for LTSs and .net for Petri nets. Adaptor interacts with two other external tools, namely TINA and CADP. TINA [29] is a tool to design and validate Petri nets. It allows to apply structural and reachability analysis on Petri nets. TINA is used in Adaptor to compute marking graphs from Petri nets encodings. CADP [21] is a toolbox to verify concurrent systems. It is used to compute the mismatch test using its EXP.OPEN tool, and to perform reductions of the adaptor LTSs using BCG_MIN and Reductor.

The current version of Adaptor fully supports transactional components. For non-transactional ones, avoiding state explosion when computing marking graphs requires that messages cannot be infinitely generated. This means first that a component should not send some message infinitely and independently (*i.e.*, without having this action triggered by a message reception or requiring an acknowledgement). In the same way, the adaptor should not infinitely and independently generate messages using vectors such as $\langle \_, \ldots, c{:}m?, \_, \ldots \rangle$.

Adaptor has been used to generate the adaptor protocols presented in this article but it has been validated and applied to many other examples as well (approximately 70 examples which correspond to 25,000 lines of XML specification) such as a Video-On-Demand service, a pervasive music player, a library lending system, and several simpler client/server systems. More details are available on the Adaptor Web page [27].

We show in Figure 16 three screenshots of Adaptor to give a flavour of what the tool looks like, here applied to eMuseum. The Adaptor GUI is made up of three different windows: the left-hand side window contains the already loaded component interfaces and contracts, the right-hand side window is used to visualise all the elements involved in the adaptation process (interfaces, contracts, Petri nets, adaptors) under different formats (graphical, textual, XML), and the bottom window is the console window. The first screenshot in Figure 16 shows the SUB

[1]approx. 5,000 lines of code correspond to the encoding of the adaptation techniques, and approx. 4,000 lines to the user interface.

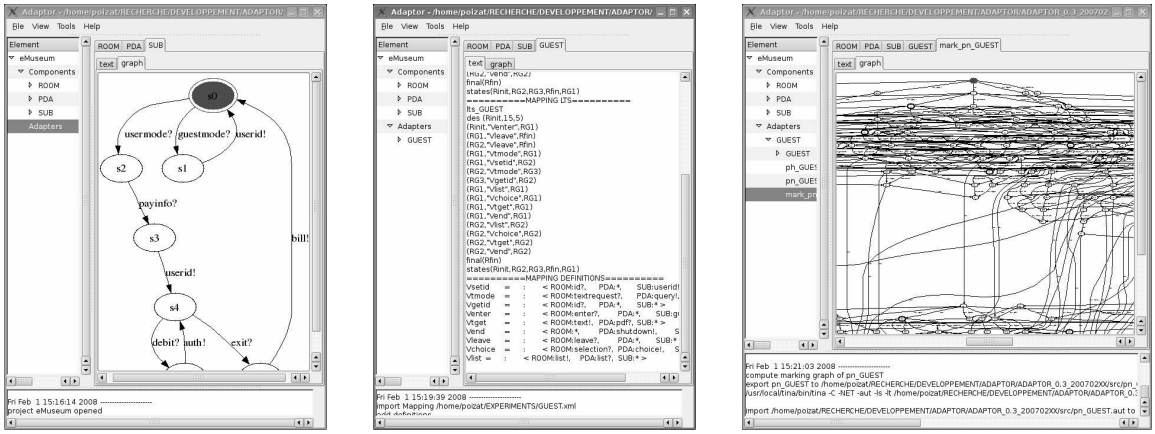


Fig. 16. Screenshots of the Adaptor tool – eMuseum, version 3

component LTS. The second screenshot is a textual description of the contract we have presented in Example 6. Finally, the last one shows a piece of the adaptor during the adaptation process (before reduction).

VII. Related Work

Software composition and adaptation is currently a hot topic in Software Engineering research. A quick look over the Web will easily produce a great number of works —ranging from deep theoretical works (*e.g.*, [30], which uses category theory for signature adaptation via *superposition*, or name morphisms) to more practical proposals (*e.g.*, [31] for Web Services). Furthermore, an increasing number of events are specifically focused on adaptation, or have it as one of their main topics (*e.g.*, the WCAT series of workshops [32], starting in 2004).

The issues related with software component integration have been a classical field of study in Software Engineering, and component mismatch has been described at all the levels of interoperability. A taxonomy of interface mismatches appears in [2], classifying them into *technical* mismatch, coming from the use of different operating systems, platforms and frameworks; *signature* mismatch, related with different names of methods and services, parameter and exception types, and parameter ordering; *protocol* or *behavioural* mismatch, caused by different message ordering, and absence or surplus of messages; *quality of service* mismatch, linked to different assumptions on properties like security, persistency, reliability or efficiency; and finally, *conceptual* or *semantic* mismatch, coming from the use of homonyms, synonyms for describing the services provided, or the existence of sub- and super-ordination relations between services.

Although some practical issues related with technical interoperability between different platforms still remain, we consider that these are not demanding a significant research effort. Accordingly, the research in the field has recently begun to explore the rest of sources of mismatch mentioned above. In particular, in this work the focus on both the signature and behavioural levels, where the use of formal notations based on logic formulas, Petri nets, process algebras, state machines, and many others have been promoted for enhancing software interfaces with a behavioural description (see [33] for an early instance). One of the first proposals for defining behavioural mismatch

from a formal point of view can be found in [8], where process algebra is used for specifying and reasoning about software composition. The work is continued in [34], where a means of characterizing connector wrappers as protocol transformations, and reasoning about their properties is presented. A similar approach is presented in [13], where compatibility and substitutability is defined in the context of CORBA, as a first attempt to put these ideas into industrial practice.

However, once behavioural mismatch is detected, the issue of how to adapt component protocols in order to solve it arises. Many of the approaches found in the literature work at the implementation level, some of them [35]–[37], related with existing programming languages and platforms, such as BPEL or SCA components, and suggesting manual or at most semi-automated techniques for solving behavioural mismatch. For instance, [35] describes a model-based approach to verifying Web service compositions, including the verification of properties created from design specifications and implementation models to confirm expected results. However, once a violation of the properties is detected, it should be manually corrected, either in the implementation of the components or in the specification models, as part of an iterative development process. Also in the context of Web services and BPEL, [36] outlines a methodology for the generation of adaptors capable of solving behavioural mismatches between BPEL processes. In their adaptation methodology, the authors use an intermediate workflow language for describing component behavioural interfaces, and they use lock analysis techniques to detect behavioural mismatch. Similarly, [37] provides automated support for the identification of protocol-level mismatches, but is able to generate an adaptor only in the absence of deadlock. If deadlock may arise from the combination of the components, the authors propose a way to handle the situation by generating a tree for all mismatches that result in a deadlock, and suggesting some hints for assisting the designer in the manual implementation of the actual adaptor.

Current approaches aiming to provide a fully automated solution to this problem are comparatively fewer, and can be divided into *restrictive*, *generative*, and *ad hoc* [4]. Restrictive approaches [38]–[42] simply try to solve the problem by cutting off the behaviour that may lead to mismatch, thus restricting the functionality of the components involved. On the contrary, generative approaches like [9], [43], [44] try to accommodate the protocols without restricting the behaviour of the components, by generating adaptors that act as mediators, remembering and reordering events and data when necessary. Finally, *ad hoc* approaches (see for instance [45]–[47]), do not address the adaptation from a general, automatable point of view, but propose specific practical solutions for particular situations instead.

The foundation for automatic behavioural adaptation was set by Yellin and Strom (YS). In their seminal article [9], they introduced formally the notion of *adaptor* as a software entity capable of enabling the interoperation of two components with mismatching behaviour. They used finite state machines to specify component interaction, to define a relation of compatibility, and to address the task of (semi-)automatic adaptor construction following the generative approach mentioned above.

More recently, Schmidt and Reussner (SR) presented a particular adaptation approach as a solution to synchronisation problems between concurrent components [45]. The proposal addresses for instance situations where one component is accessed simultaneously by two other components. The approach is based on algorithms close to the

synchronous products we use in this article. Moreover, they can solve protocol incompatibilities enabling one of the involved components to perform several communication actions before or after synchronising with its partners. These ideas are implemented in the CoConut/J tool suite [48], where the authors introduce the concept of parameterised contracts and a model for component interfaces. The paper also presents algorithms and tools for specifying and analysing component interfaces in order to check interoperability and to generate adapted component interfaces automatically. In comparison, our proposal is more general and based on a rich notation to deal with possibly complex adaptation scenarios, whereas the SR approach works out only precise situations in which mismatch may happen, without using any contract language for adaptor specification.

In their paper *Adapt or Perish* [49], Dumas and collaborators presented an approach to behavioural interface adaptation based on the definition of a set of adaptation operations for establishing the basic relation patterns between the messages names used in the components being adapted, and they defined a trace-based algebra for describing the transformations required to solve the adaptation problem. They also present a visual notation for describing a mapping between the behavioural interfaces of the components. Their approach is similar to ours in the sense that these basic operations correspond to the different relations (one-to-one, one-to-many, many-to-one, one-to-zero, etc) between message names that can be defined by means of our synchronous vectors. However, their proposal does not present a solution for deriving an adaptor from the visual mappings, but just contains a preliminary (*i.e.*, non sufficient) condition for detecting deadlock scenarios in the behavioural interfaces. Moreover, their mappings require to relate the messages at the behavioural level (*i.e.*, matching messages directly from the component protocol specifications), while our adaptation contracts are more abstract, since the mapping is performed at the signature level (*i.e.*, between the messages declared in the component interfaces) from which we automatically obtain an adaptor solving the mismatch at the behavioural level. Finally, their approach is not able to perform message reordering when it is required for solving the problem.

Taking the YS proposal as a starting point, the work of Brogi and collaborators (BBCP) [43], [44] presents a methodology for generative behavioural adaptation. In their proposal, component behaviour is specified using a process algebra —a subset of the π-calculus—, where service offering/invocation is represented by input/output events in the calculus, respectively. The starting point of their adaptation process is a *mapping*, an adaptation contract that states correspondences between the services of the components being adapted. Then, an adaptor generation algorithm refines the specification given by the mapping into a concrete adaptor implementation, taking into account the behavioural interfaces of the components, which ensures correct interaction between them according to the mapping. The adaptor is able to accommodate not only signature mismatch between service names, but also behavioural mismatch (*i.e.*, the interaction protocols that the components follow, or the partial ordering in which services are offered/invoked).

Another interesting proposal in this field is that of Inverardi and Tivoli (IT) [38]. Certain aspects of their work go beyond BBCP by addressing how to enforce certain behavioural properties (namely liveness and safety properties expressed as specific processes) out of a set of already implemented behaviours. Starting from the specification with MSCs of the components to be assembled and of the properties that the resulting system should verify, they

automatically derive the adaptor glue code for the set of components in order to obtain a property-satisfying system. In order to do that, they follow the so-called restrictive approach. The IT proposal was extended in [39] with the use of temporal logic; coordination policies are expressed as LTL properties, and then translated into Büchi automata. Recent outcomes of this research line allow a distributed implementation of the adaptors [40], and take into account time and other QoS issues [41].

Another example of the restrictive approach is the work of de Alfaro and collaborators [11], [42], who use game theory to achieve behavioural adaptation. One of the relevant features of the proposal is that time information can be taken into account within the component interfaces.

Our approach to behavioural adaptation can be considered as *both generative and restrictive*, since we address behavioural adaptation by enabling event reordering (as in BBCP), while we also remove incorrect behaviour (as in IT). Similarly to both of them, our main goal is to ensure deadlock freedom. However, more complex adaptation policies and properties can be specified by means of our vector LTSs. A deeper comparison with the aforementioned approaches yields that our proposal addresses system-wide adaptation (*i.e.*, differently from BBCP, it may involve more than two components), and that it is based on LTS descriptions of component behaviour, instead of using process algebra as in BBCP. However, we may also describe behaviour by means of a simple process algebra, and use its operational semantics to derive LTSs from it [14]. Differently from IT, which requires name matching, we use synchronous vectors in our adaptation contracts, playing a similar function than the mapping rules in BBCP. With that, we are able to perform adaptation of incompatible events. Finally, our approach is fully tool equipped, while BBCP have only presented a sketch of the implementation of their adaptation algorithm.

Nevertheless, the most relevant achievement of our current proposal is the use of vector LTSs for imposing additional properties over adaptation contracts. In fact, the semantics of BBCP mappings can be expressed by combining their different rules in a vector LTS with a single state and all vector transitions looping on it. On the contrary, our vector LTSs are much more expressive, solving the problem of BBCP underspecified mappings [43], and allowing to take into account a new class of adaptation problems.

A different characterisation of behavioural adaptation techniques may classify them into *immutable* and *contextual*. Immutable approaches are those that define a static set of rules for describing the adaptation required, and these rules are applied uniformly during the whole adaptation process. On the contrary, contextual adaptation pays attention to context information in order to decide on-the-fly the adaptation strategy to apply. Our present approach allows contextual adaptation by the use of vector LTSs which govern when the adaptation rules are applied (as shown in Figs. 7 and 13), while the rest of the approaches mentioned above are static. Some recent works based on the BBCP proposal try to address more flexible ways of contextual adaptation [50].

Finally, most of the current adaptation proposals — and our present work among them— may be considered as *global*, since they proceed by computing global adaptors for closed systems made up of a predefined and fixed set of components. However, this is not satisfactory when the system may evolve, with components entering or leaving it at any time, *e.g.*, for pervasive computing. To enable adaptation on such systems, an *incremental* approach should be considered, by which the adaptation is dynamically reconfigured depending on the components present in the

system. One of the first attempts in this direction is [51], whose proposal for incremental software construction by means of refinement allows for simple signature adaptation. However, to our knowledge the only proposal addressing incremental adaptation at the behavioural level is [52].

VIII. Concluding Remarks

Software Adaptation is widely accepted as a promising solution to favour the reuse of black-box components that require non intrusive adjustments to make them fit with the specificities of the system-to-be. In this article, we have presented a proposal for software adaptation at the signature and behavioural levels based on a simple adaptation contract notation. These contracts can be used to express correspondences (possibly involving mismatching messages) between an arbitrary number of components, but also complex adaptation scenarios. Our proposal is equipped with two algorithms depending whether reordering is necessary or not in the adaptation process. The first one is based on synchronous product computation, and the second one on encodings into Petri nets. Our proposal is completely supported by a tool which was applied to many examples.

In this article we follow a regular model-based approach, focusing on abstract (platform-independent) behavioural interface models, LTSs. It has been demonstrated, usually for verification purposes, that such abstract models could be derived from existing implementation platforms' languages, *e.g.*, [53]–[55] for Web services. As regards adaptation, model-based behavioural adaptation has been applied to COM/DCOM components in [38] and to Web services in [36], [37]. In a recent paper [15], we have addressed WF components. We have shown how LTS descriptions could be extracted automatically from WF workflows, and how a new WF component could be obtained from an adaptor protocol generated with the techniques we have presented here. Therefore, we think the proposed adaptation techniques are of great interest for real-world software components or Web services.

There are still some open issues in our proposal deserving future work. In this part of the conclusion, we will particularly emphasise three perspectives, namely data adaptation, contract design support, and application to pervasive systems.

Data adaptation. Taking data exchange into account in protocols is important to ensure full compatibility. So far, this can be supported in the approach at hand using additional messages for data exchange in the abstract component protocols (LTSs), as presented in Section II-A. Provided this encoding is performed as a pre-processing, and the adaptation contract takes the additional messages into account, the protocols can be adapted, as demonstrated in [15].

Supporting directly data types would be more efficient but would require first more expressive models than LTSs. In particular, we consider Symbolic Transition Systems (STSs) [56] or Extended State Diagrams [57] as good candidates since they allow the description of the data involved in the operations within the protocol without suffering from the state explosion problem. Then, data types should be taken into account also in the contract specification as for the additional message encoding technique, above. As far as the adaptation process itself is concerned, we are studying two possible techniques. The first one is compatible with the approach at hand, *e.g.*, for the reordering approach, it consists in taking the data types into account in the Petri net encoding patterns (data types resources being generated for component emissions, data type resources being consumed for components

receptions and data types being transferred for data vectors). We are currently looking for an efficient Petri net encoding, using Petri net extensions, in order to avoid state explosion problems. The second technique specifically addresses these efficiency issues. It consists in implementing data adaptation separately from the message-based one, through a data adaptation engine that would be embedded in the adaptor implementation, and that would store received values and redistribute them *wrt.* the correspondences expressed in the data contract.

Support for contract design. The design of an adaptation contract may be a non-trivial and error-prone task, leading to too many interactions being removed in the adaptation process to ensure deadlock-freedom. To address this issue, recent work has focused on post-generation adaptor assessment, either by reusing existing model-checkers [58], or by developing new tools such as Clint [59], that is able to graphically represent deadlocks in components and interactions that are removed in the adaptation process. The former approach is more powerful yet it requires temporal logic formulas are given. This is the approach we have used in this article to obtain our mappings. The later is less expressive (as far as the kind of properties which are assessed over the adapted system are concerned) yet, it benefits from being fully-automatic.

Approaches dedicated to the automatic generation of compositions are indeed the current goal of research groups working at the semantic interoperability level, *e.g.*, adding semantic annotation to (Web) services [60]. Yet, enforcing a semantic description for all components (including legacy ones) is a strong assumption.

We are convinced that an assisted design approach is a good trade-off between complete automation and manual writing of the composition and adaptation contracts. Further, it enables a end-user composition vision [61], [62]. As a perspective, we plan to propose techniques to support the contract design task. A partial specification of the contract could be given for which remaining composition issues (such as deadlocks in components and interactions that would be removed by the adaptation process) would be emphasised using Clint. In addition, incremental contract construction, where at each step possible message correspondences to complete the contract would be proposed, would foster the user-friendliness of the contract design process.

Self-adaptive pervasive systems. A perspective in the context of funded research projects is to apply our adaptation techniques to pervasive systems. In this field, self-adaptation is a mandatory feature because less assumptions can be done on the system at hand, *e.g.*, new components or services can show up or disappear at run-time while the overall adaptation mechanism should support these evolutions and keep on making the system work in a reliable way. Dynamic Aspect Oriented Programming is a technology we are currently exploring as well to put into practice adaptation techniques in this highly dynamic context.

Acknowledgements

The authors thank Sandrine Beauche and Juan David G. Urbano for their participation to the implementation of Adaptor, as well as Bernard Berthomieu, Frédéric Lang, Massimo Tivoli, and François Vernadat for their help on external tool support, interesting comments and fruitful discussions. The authors are also grateful to the anonymous referees whose comments helped a lot to improve this article. This work is partially supported by the

projects “PERvasive Service cOmposition” (PERSO) funded by the French National Agency for Research (ANR-07-JCJC-0155-01), TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and P06-TIC-02250 funded by the Andalusian local Government.

REFERENCES

[1] O. Nierstrasz and T. D. Meijler, “Research Directions in Software Composition,” *ACM Computing Surveys*, vol. 27, no. 2, pp. 262–264, 1995.

[2] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, “Towards an Engineering Approach to Component Adaptation,” in *Architecting Systems with Trustworthy Components*, ser. Lecture Notes in Computer Science, vol. 3938. Springer, 2006, pp. 193–215.

[3] C. Canal, J. M. Murillo, and P. Poizat, “Coordination and Adaptation Techniques for Software Entities,” in *ECOOP 2004 Workshop Reader*, ser. Lecture Notes in Computer Science, vol. 3344. Springer, 2004, pp. 133–147.

[4] ——, “Software Adaptation,” *L’Objet. Special Issue on Coordination and Adaptation Techniques*, vol. 12, no. 1, pp. 9–31, 2006.

[5] G. Agha, “Special Issue on Adaptive Middleware,” *Communications of the ACM*, vol. 45, no. 6, pp. 30–64, 2002.

[6] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[7] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, “Making Components Contract Aware,” *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[8] R. Allen and D. Garlan, “A Formal Basis for Architectural Connection,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

[9] D. M. Yellin and R. E. Strom, “Protocol Specifications and Components Adaptors,” *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[10] J. Magee, J. Kramer, and D. Giannakopoulou, “Behaviour Analysis of Software Architectures,” in *Proc. of the 1st Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic Publishers, 1999, pp. 35–49.

[11] L. de Alfaro and T. A. Henzinger, “Interface Automata,” in *Proc. of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE’01)*. ACM Press, 2001, pp. 109–120.

[12] F. Plasil and S. Visnovsky, “Behavior Protocols for Software Components,” *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, 2002.

[13] C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo, “Adding Roles to CORBA Objects,” *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 242–260, 2003.

[14] C. Canal, P. Poizat, and G. Salaün, “Synchronizing Behavioural Mismatch in Software Composition,” in *Proc. of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS’06)*, ser. Lecture Notes in Computer Science, vol. 4037. Springer, 2006, pp. 63–77.

[15] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat, “A Model-Based Approach to the Verification and Adaptation of WF/.NET Components,” in *Proc. of the 4th International Workshop on Formal Aspects of Component Software (FACS’07)*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2007, to appear.

[16] K. Scribner, *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.

[17] J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds., *Handbook of Process Algebra*. Elsevier, 2001.

[18] R. Milner, *Communication and Concurrency*, ser. International Series in Computer Science. Prentice Hall, 1994.

[19] M. Bernardo and P. Inverardi, Eds., *Formal Methods for Software Architectures*, ser. LNCS. Springer, 2003, vol. 2804.

[20] A. Arnold, *Finite Transition Systems*, ser. International Series in Computer Science. Prentice Hall, 1994.

[21] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, “CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes,” in *Proc. of 19th International Conference on Computer Aided Verification (CAV’07)*, ser. Lecture Notes in Computer Science, vol. 4590. Springer, 2007, pp. 158–163.

[22] S. Haddad and P. Poizat, “Transactional Reduction of Component Compositions,” in *Proc. of the 27th IFIP International Conference on Formal Methods for Networked and Distributed Systems (FORTE’07)*, ser. Lecture Notes in Computer Science, vol. 4574. Springer, 2007, pp. 341–357.

[23] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[24] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioural Models from Scenarios," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 99–115, 2003.

[25] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[26] C. Rackoff, "The Covering and Boundedness Problems for Vector Addition Systems," *Theoretical Computer Science*, vol. 6, pp. 223–231, 1978.

[27] "Adaptor, December 2007 distribution (LGPL licence)." 2007, `http://adaptorweb.lcc.uma.es/`.

[28] E. Gansner, E. Koutsofios, and S. North, *DOT User's Manual*, Jan. 2006.

[29] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The Tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets," *International Journal of Production Research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[30] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A Graph Based Architectural (Re)configuration Language," in *Proc. of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'01)*. ACM Press, 2001, pp. 20–32.

[31] S. Dustdar and W. Schreiner, "A Survey on Web Services Composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.

[32] "International Workshop Series on Coordination and Adaptation Techniques," `http://wcat.unex.es`.

[33] D. Lea and J. Marlowe, "Interface-Based Protocol Specification of Open Systems Using PSL," in *Proc. of the 9th European Conference Object-Oriented Programming (ECOOP'95)*, ser. Lecture Notes in Computer Science, vol. 952. Springer, 1995, pp. 374–398.

[34] B. Spitznagel and D. Garlan, "A Compositional Formalization of Connector Wrappers," in *Proc. of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 2003, pp. 374–384.

[35] H. Foster, S. Uchitel, J. Maggee, and J. Kramer, "Model-based Verification of Web Service Compositions," in *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*. IEEE Computer Society, 2003, pp. 152–163.

[36] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," in *Proc. of the 4th International Conference on Service Oriented Computing (ICSOC'06)*, ser. Lecture Notes in Computer Science, vol. 4294. Springer, 2006, pp. 27–39.

[37] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," in *Proc. of the 16th International World-Wide Web Conference (WWW'07)*, 2007, pp. 993–1002.

[38] P. Inverardi and M. Tivoli, "Deadlock Free Software Architectures for COM/DCOM Applications," *Journal of Systems and Software*, vol. 65, no. 3, pp. 173–183, 2003.

[39] ——, "Software Architecture for Correct Components Assembly," in *[19]*, pp. 92–121.

[40] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli, "SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems," in *Proc. of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, 2007, pp. 784–787.

[41] M. Tivoli, P. Fradet, A. Girault, and G. Goessler, "Adaptor Synthesis for Real-Rime Components," in *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, ser. Lecture Notes in Computer Science, vol. 4424. Springer, 2007, pp. 185–200.

[42] L. de Alfaro and M. Stoelinga, "Interfaces: A Game-Theoretic Framework to Reason about Open-Systems," in *Proc. of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03)*, ser. Electronic Notes in Theoretical Computer Science, vol. 97, 2004, pp. 3–23.

[43] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *Journal of Systems and Software*, vol. 74, no. 1, pp. 45–54, 2005.

[44] A. Brogi, C. Canal, and E. Pimentel, "Component Adaptation Through Flexible Subservicing," *Science of Computer Programming*, vol. 63, no. 1, pp. 39–56, 2006.

[45] H. W. Schmidt and R. H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronization," in *Proc. of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*. Kluwer Academic Publishers, 2002, pp. 213–229.

[46] H. Min, S. Choi, and S. Kim, "Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition," in *Proc. of 7th International Symposium on Component-Based Software Engineering (CBSE'04)*, ser. Lecture Notes in Computer Science, vol. 3054. Springer, 2004, pp. 40–47.

[47] B. Benatallah, F. Casati, D. Grigori, H. R. Motahari-Nezhad, and F. Toumani, "Developing Adapters for Web Services Integration," in *Proc. of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3520, pp. 415–429.

[48] R. H. Reussner, "Automatic Component Protocol Adaptation with the CoConut/J Tool Suite," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 627–639, 2003.

[49] M. Dumas, K. W. S. Wang, and M. L. Spork, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," in *Proc. of the 4th International Conference on Business Process Management (BPM'06)*, ser. Lecture Notes in Computer Science, vol. 4102. Springer, 2006, pp. 65–80.

[50] J. Cubo, G. Salaün, J. Cámara, C. Canal, and E. Pimentel, "Context-Based Adaptation of Component Behavioural Interfaces," in *Proc. of the 9th Conference on Coordination Models and Languages (Coordination'07)*, ser. Lecture Notes in Computer Science, vol. 4467. Springer, 2007, pp. 305–323.

[51] R. J. Back, "Incremental Software Construction with Refinement Diagrams," Turku Center for Computer Science, Tech. Rep. 660, 2005.

[52] P. Poizat and G. Salaün, "Adaptation of Open Component-based Systems," in *Proc. of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, ser. Lecture Notes in Computer Science, vol. 4468. Springer, 2007, pp. 141–156.

[53] X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services," in *Proc. of the 13th International Conference on World Wide Web (WWW'04)*. ACM Press, 2004, pp. 621–630.

[54] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Algebra," in *Proc. of the IEEE International Conference on Web Services (ICWS'04)*. IEEE Computer Society, 2004, pp. 43–51.

[55] H. Foster, S. Uchitel, and J. Kramer, "LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography," in *Proc. of 28th International Conference on Software Engineering (ICSE'06)*. ACM Press, 2006, pp. 771–774.

[56] O. Maréchal, P. Poizat, and J.-C. Royer, "Checking Asynchronously Communicating Components using Symbolic Transition Systems," in *Proc. of the International Symposium on Distributed Objects and Applications (DOA'2004)*, ser. Lecture Notes in Computer Science, vol. 3291. Springer, 2004, pp. 1502–1519.

[57] C. Attiogbé, P. Poizat, and G. Salaün, "A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes," *IEEE Transactions on Software Engineering*, vol. 33, no. 3, pp. 157–170, 2007.

[58] P. Poizat, G. Salaün, and M. Tivoli, "An Adaptation-based Approach to Incrementally Build Component Systems," in *Proc. of the 3rd International Workshop on Formal Aspects of Component Software (FACS'06)*, ser. Electronic Notes in Theoretical Computer Science, no. 182. Elsevier, 2007, pp. 155–170.

[59] J. Cámara, G. Salaün, and C. Canal, "Clint: A Composition Language Interpreter," in *Proc. of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08)*. Springer, 2008, to appear.

[60] S. Ben Mokhtar, N. Georgantas, and V. Issarny, "COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support," *Journal of Systems and Software, Special Issue on ICPS'06*, vol. 80, no. 12, pp. 1941–1955, 2007.

[61] M. Burnett, C. Cook, and G. Rothermel, "End-user software engineering," *Communications of the ACM*, vol. 47, no. 9, pp. 53–58, 2004.

[62] X. Liu, G. Huang, and H. Mei, "Towards End User Service Composition," in *Proc. of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, 2007, pp. 676–678.

APPENDIX I
ADDITIONAL FIGURES

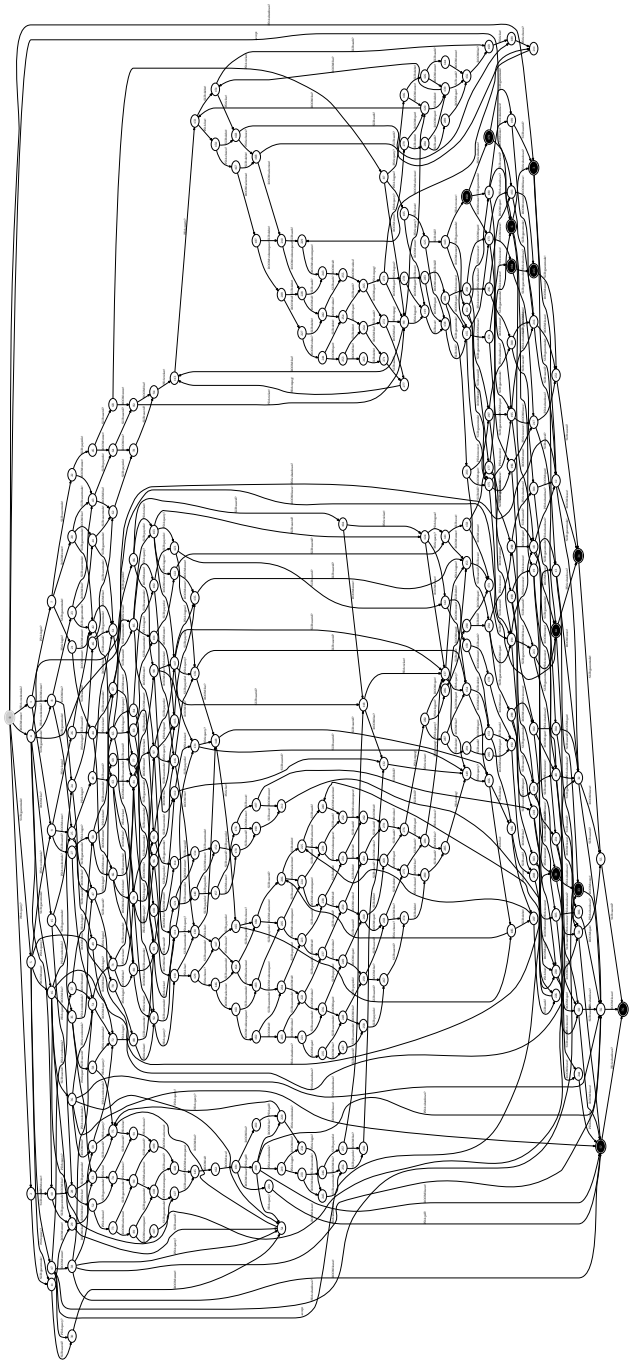


Fig. 17. Adaptor protocol for eMuseum, version 3 (full mode)

II. PROOFS

Proof: [Alg. 1 correctness] It follows from Def. 9 that the set of traces of the adaptor LTS resulting from the product, $\Pi_L((C_1, \ldots, C_n), L)$, contains all interactions which are possible in between the components. The reason is that the product is computed taking into account, at each moment, which messages are available for emission or reception in the different components, and keeps in the product only the transitions corresponding to legal correspondences defined in the vector LTS. Thereafter, the application of *remove_deadlocks* ensures all interactions ending in incorrect states (deadlocks) are removed. It therefore results from the two points above that only correct interactions, *i.e.*, such that (i) they ensure termination in final states and (ii) they correspond at each step to messages that are sent or received by components, are preserved.

Then, directions of events are reversed. This ensures the adaptor resulting from it, let us note it A_L, can communicate correctly at each step of the aforementioned traces with the components: the product $A_L||\Pi_L((C_1, \ldots, C_n), L)$ where synchronisation is made on a vector basis – LTS labels are vectors in A_L and $\Pi_L((C_1, \ldots, C_n), L)$, and are synchronised if they correspond – is correct. Permutations then replace in A_L each transition labelled with a vector by a sub-LTS whose set of traces corresponds to all possible event orderings of the vector, yielding adaptor Ad. Therefore, for any order in which the different components implied in a step of a vector trace do communicate – *i.e.* for all possible component communications interleavings ($|||$) – the adaptor is ready to communicate on the corresponding communication event. This yields $Ad||(C_1||| \ldots |||C_n)$ is correct. Moreover, since prefixing is used, components in $C_1|| \ldots ||C_n$ do not synchronise and therefore it follows that $Ad||(C_1||| \ldots |||C_n)$ is equivalent to $Ad||(C_1|| \ldots ||C_n)$, and therefore Ad is correct. Note that for optimising reasons, event reversal and permutations are performed at the same time in the algorithm. Adaptation being a process which is dependent of the adaptation contract, the process may yield empty adaptors in some cases, since, as in all restrictive adaptation approaches (see related work in Section VII), removal of paths to error states (here deadlocks) may reduce the set of correct interactions to none. Then, putting such an adaptor in the component system will yield also an adapted system in which no interactions are possible. Making initial states being final ($I \in F$, Def. 1) ensures this is correct. ■

Proof: [Alg. 3 correctness] The Petri net based adaptor computation relies on the encoding of different parts relative to the components interfaces with event mirroring (let us note them PN_i, $i \in \{1, \ldots, n\}$) and to the vectors and vector LTS (let us note it PN_L). Taking each PN_i separately, and supposing the places corresponding to messages sent by the adaptor (the $!!c_i : a$ places) are always fed, then its marking graph, $M(PN_i)$[2], is an LTS which exactly corresponds (through mirroring) to component C_i LTS. Therefore, all $M(PN_i)||C_i$ are correct (no deadlock, components end in final states). Taking the product of these LTS, $M(PN_1)|| \ldots ||M(PN_n)$, yields a perfect adaptor – $(M(PN_1)|| \ldots ||M(PN_n))||C_1|| \ldots ||C_n$ has no deadlock – as (i) $(M(PN_1)|| \ldots ||M(PN_n))||C_1|| \ldots ||C_n$ is equivalent to $(M(PN_1)||C_1)|| \ldots ||(M(PN_n)||C_n)$ thanks to indexing which ensures event names disjointness, and (ii) each $M(PN_i)||C_i$ is correct.

[2] M corresponds to the *get_marking_graph* function in Algorithm 3, and is used here for the sake of conciseness.

Now, taking the global Petri net marking graph (let us note it Ad), where separate nets are glued using PN_L, we can observe that the set of traces of Ad is, up to graph reduction (see comment below), a subset of the set of traces of $M(PN_1)||\ldots||M(PN_n)$. This results from the fact that now the input places are only fed through place transfers defined in PN_L, *e.g.*, some transition $c_i : a!$, requiring a token in place $!!c_i : a$, is only possible now if (provided that a vector $\langle c_i : a?, c_j : b! \rangle$ exists) first, transition $c_j : b?$ has been fired – adding a token in place $??c_j : b$ – and then, a transfer from place $??c_j : b$ to place $!!c_i : a$ has been done – using a tau transition. Yet, all remaining traces of Ad (with respect to $M(PN_1)||\ldots||M(PN_n))$ are correct as each such trace either:

(i) ends in a final state and, being also up to graph reduction a trace in $M(PN_1)||\ldots||M(PN_n)$, results in correct interactions with the components (mirroring and ordering), or

(ii) does not end in a final state and hence is removed by the *remove_deadlocks* step.

The reduction step is performed at the end of the adaptor computation process. As this reduction respects deadlock freedom (in the usual acceptation of it, *i.e.*, there is no state in S without outgoing transitions), and since our deadlock freedom property is a weaker property (there is no state in $S \backslash F$ without outgoing transitions), it results that reduction respects the adaptor properties. ■